)OCUMENT PREPARATION SYSTEMS
'. Nievergelt, G. Coray, J.D. Nicoud, A.C. Shaw (eds.)
North-Holland Publishing Company, 1982

Interactive Editing Systems

Norman Meyrowitz
Andries van Dam

Department of Computer Science
Box 1910
Brown University
Providence, Rhode Island 02912

ABSTRACT

Many daily tasks, whether done with conventional tools or with computers, can be viewed as **editing** tasks: tasks in which the state of some target entity is changed by the user. This article examines computer-based **interactive editing systems**, which allow users to change the state of targets such as manuscripts and programs.

We present the material in two parts: a tutorial and a survey. The tutorial defines terms and introduces issues for the novice, and provides a reference for the more knowledgeable reader. Our aim is to provide a comprehensive and systematic view of the features of typical systems, highlighting substantive similarities and differences. We provide user and system views of the editing process, present an historical perspective, and cover the functional capabilities of editors, emphasizing user-level rather than implementation-level considerations.

The survey is intended for a more general audience: the more experienced user and the editor-designer as well as the curious novice. It presents numerous examples of the state-of-the-art in both the academic and commercial arenas, covering line editors, screen editors, interactive editor/formatters, structure editors, syntax-directed editors, and commercial word processing editors. We discuss pertinent issues in the field, and conclude with some observations about the future of interactive editing.

CR Categories and Subject Descriptors: I.7.1 [**Text Processing**]: Text Editing—*languages, spelling*; I.7.0 [**Text Processing**]: General; I.7.2 [**Text Processing**]: Document Preparation—*format and notation, languages, photocomposition*; I.7.4 [**Text Processing**]: Miscellaneous; H.4.1 [**Information Systems Applications**]: Office Automation—*equipment, word processing*; D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; D.2.3 [**Software Engineering**]: Coding—*program editors, pretty printers.*

General Terms: Human Factors, Languages, Design.

Additional Keywords and Phrases: editing, editors, text editing, text editors, program editors, syntax-directed editors, structure editors, user interface design, text formatting, text processing, word processing.

Norman Meyrowitz, Andries van Dam. Interactive Editing Systems. *Computing Surveys*, Vol. 14, No. 3, Sept. 1982, 'Copyright 1982 Association for Computing Machinery, Inc. Reprinted by permission.'

INTRODUCTION

The interactive editor[1] has become an essential component of any computing environment. It uses the power of the computer for the creation, addition, deletion, and modification of text material such as program statements, manuscript text, and numeric data. The editor allows text to be modified and corrected many orders of magnitude faster and easier than would manual correction.

Though editors have always been deemed important tools in computing systems, they have only recently become a fashionable topic of research, as they become key components in the office of the future. No longer are editors thought of as tools only for programmers, or for secretaries transcribing from marked-up hardcopy generated by authors. It is now increasingly realized that the editor should be considered the primary interface to the computer for all types of **knowledge workers**, as they compose, organize, study, and manipulate computer-based information. Unlike the literature in areas such as programming languages or operating systems (with rich collections of written materials, from basic definitions and tutorials to complex formalisms, analyses, and implementation strategies), the literature in the field of editing consists primarily of functional descriptions of particular editors. Little work has been done to standardize terminology or to create a framework for comparing, contrasting, and analyzing editing systems.

This article is designed as a starting point for developing such a framework. Our intended audience is composed of two distinct groups: those with only limited experience with an interactive editor and those with a broader background in the field. Part I of the paper is tutorial in nature and is meant for those with only superficial experience in using an editor to create program or document text. It is a reasonably comprehensive introduction to text editing. As such, it is not meant to be read in its entirety by the editor designer, but rather to be used as a reference. It is here that we present our definitions and overview of the field, making explicit what has previously been largely implicit or passed on as "oral tradition." We look at document editing from a user and system standpoint, at some of the major historical developments, and at the functional capabilities of editors.

Part II presents technical details of specific editors, using the terminology and concepts laid out in Part I. It is intended for a broader audience, including those quite familiar with the concepts covered in the first half as well as those comfortable with the editors in their own computing environments but not necessarily familiar with the range of editors available. This part surveys editors available in the academic and commercial realms, providing points of departure for further investigation rather than an exhaustive point-by-point comparison. We discuss unresolved issues in the field, and examine the future of editing. A bibliography provides a list of materials for further reading.

Several topics directly related to interactive editing will **not** be covered here; instead, we supply references to surveys in those fields. In particular, text formatting techniques are left to Furuta et al. [FURU82], implementation of editors is covered by Rice and van Dam [RICE71] and Finseth [FINS80], editor evaluation is surveyed by Embley and Nagy [EMBL81], and graphics editors are described in Newman and Sproull [NEWM79] and Foley and van Dam [FOLE82].

[1] "Editor" throughout refers to an interactive editing program, not to an author/user correcting a document.

PART I: TUTORIAL

1. GENERAL OVERVIEW

1.1 The Editing Process

An **interactive editor**[2] is a computer program that allows a user to create and revise a target **locument**. We use the term "document" to include targets such as computer programs, text, equations, tables, diagrams, line art, and halftone or color photographs – anything one might find on a printed page. In this paper, we restrict our discussion to **text editors** (in which the primary elements being manipulated are character strings of the target text) used for manuscript and program production, and to **structure editors** (in which the primary elements being manipulated are portions of some generic structure such as a tree).

The document-editing process is an interactive user-computer dialogue 1) to select what part of the target is to be viewed and manipulated, 2) to determine how to format this view online and then to display it, 3) to specify and execute operations that modify the target document, and 4) to update the view appropriately.

Selection of the part of the document to be viewed and edited involves first **traveling** through the document to locate the area of interest with operations such as `next screenful`, `bottom`, and `find pattern`. Next, having specified *where* the area of interest is, the selection of *what* is to be viewed and manipulated there is controlled by **filtering**. Filtering extracts the relevant subset of the target document at the point of interest, such as the next screen's worth of text or the next statement. **Formatting** then determines *how* the result of the filtering will be seen as a visible representation, the **view**, on a display screen or hardcopy device.

Only in the actual **editing** phase is the target document created or altered per a set of operations, commonly including `insert`, `delete`, `replace`, `move`, and `copy`. The editing functions are often specialized to operate on **elements** meaningful to the type of editor, such as single characters, words, lines, sentences and paragraphs for manuscript-oriented editors, and keywords and source language statements for program-oriented editors.

In a simple scenario, then, the user might **travel** to the end of a document. A screen's worth of text would be **filtered**, and this subset would be **formatted** and the **view** displayed on an output device. The user could then, for example, `delete` the first three words of this view.

1.2 The Editor: A User Viewpoint

The user of an interactive editor is presented with a **conceptual model** of the system, which is the designer's abstract framework on which the editor and the "world" in which it operates are based, and with a **user interface**, the collection of tools and techniques with which the user communicates with the editor.

The conceptual model, in essence, provides an easily understood abstraction of the target document and its elements, and a set of guidelines with which to anticipate the effects of operations on these elements. Conceptual models range from those that are hardly visible to the user and not very cohesive or thorough, to those that are well-articulated and provide a consistent and complete framework both for using and implementing the system. In others, the conceptual model is incomplete; it is insufficient to describe more than a very cursory notion of the system. Some of the early **line editors** simulated the world of the keypunch, allowing

[2] In this paper, Bold type is used to introduce concepts and terms. Typewriter type is used to set off editor commands. Italic type is used for emphasis.

operations upon numbered sequences of 80-character card-image lines, either within a single line or upon an integral number of lines. Some more modern **screen editors** define a world in which a document is represented as a quarter-plane of text lines, unbounded both downward and to the right. The user sees through a cutout only a rectangular subset of this plane at any time on a multi-line display screen, but can move the cutout both left and right and up and down to see other portions of a document. Operations manipulate portions of this quarter-plane without regard to line boundaries.

The user interface contains the **input devices**, the **output devices**, and the **interaction language** of the system. Examples of each of these will be discussed in the following subsections. Along with the user interface, the user is often given documentation that may include a description of the conceptual model, a high-level description of the system architecture in user-level terminology, a user's guide detailing the syntax and semantics of the interaction language, and a tutorial that provides operational definitions and demonstrates typical situations with examples.

Each individual forms a personal **user model** of an editing system, partly extrapolated from information provided in the recorded documentation or passed on by "experts" and partly based on repeated use of the system. The user model may differ from the conceptual model in several ways. First, it can be thought of as a subset of the conceptual model. For example, a user experienced in document preparation may have no notion of the language-specific commands for correct program indentation; similarly, a programmer may have no notion about the features provided for online table manipulation. Second, when the user consistently uses combinations of primitives to perform common operations in ways not originally encompassed by the conceptual model, the user model can be thought of as an extension of the conceptual model. Third, the user model can be thought of as an operationally equivalent but logically different form of the conceptual model. For example, rather than considering his/her actions to be moving a cutout over various parts of a document as described above, the user may consider the cutout as stationary and the *document* as scrollable horizontally and vertically past this cutout.

The user model is a personalized, high-level understanding of the conceptual model provided, of the manipulable entities and their interrelationships, of the set of operations allowed on the entities, and of the interaction language used to invoke these operations. The user model is not simply descriptive but prescriptive as well: to perform editing tasks, the user employs the user interface based on his/her user model.

1.2.1 User Interface

1.2.1.1 Input Devices

Input devices are used for three main purposes: 1) to enter elements; 2) to enter commands; and 3) to designate editable elements. These devices, as used with editors, can be divided into three categories: **text** devices, **button** devices, and **locator** devices [GSPC79, FOLE82].

Text or **string** devices are typically typewriter-like keyboards on which a user presses and releases a key to send to the CPU or to an I/O controller a unique code for each key. Essentially all present computer keyboards are of the QWERTY variety. This variety of keyboard, named for the first six letters in the second row of the keyboard, was invented by Christopher Latham Sholes in the 1860's. The strange key layout was done for purely mechanical reasons – letters most commonly used together were placed as far apart as possible so mechanical typewriter keys would not clash. As surveyed in MONT82, several experimental text input devices have been constructed. The Dvorak Simplified Keyboard rearranges the keys to reduce fatigue and increase typing speed; despite experimental evidence showing its superiority, it has not caught on strongly. The Montgomery "wipe-activated" keyboard requires the user not to press keys,

but rather to "wipe" a stylus across the keyboard surface. Letters comprising common digrams and trigrams, such as `TH` and `ING` are placed next to one another on the keyboard; the user simply wipes the wand from left to right across the letters, rather than typing two or three separate keystrokes. NLS's keyset [ENGE68] consists of a pad of five long keys, similar in shape to white piano keys, and provides a method for entering small text additions or corrections. Each key corresponds to a position in a 5-bit binary word called a "chord"; by depressing the proper combination, the user can represent $2^5 - 1 = 31$ numbers or ASCII codes (binary "00000" is not counted, as this means no keys are being pressed).

An alternative to direct keyboard input is **optical character recognition (OCR)**. Typically a standard electric typewriter, (inexpensively) equipped with an OCR typeface, is used for typing the text on normal paper. This paper is then fed through an optical character reader, an electromechanical device which scans the page and translates each character into the proper digital representation for that computer system. This technique allows the continued use of pre-existing facilities: the typewriters become offline "links" to the computer facilities. Although many OCR systems also allow rudimentary editing of the raw text stream via `character-` and `line-delete` control characters inserted in the text, this cannot compete with the convenience and essentially instant editing facility of online input. While OCR is often seen as an inexpensive way to begin "computerization," it is contrary to the spirit of online authoring, in which the author is able to express his/her thoughts and experiment with a composition from its inception.

Button or **choice** devices generate an interrupt or set a system flag, usually causing invocation of an associated application program action. They typically are grouped as a set of special function keys on the alphanumeric keyboard or on the display itself. Alternatively, buttons are often simulated in software by having the user choose text strings or symbols displayed on a screen.

Locator devices are x-y analog-to-digital transducers that position a cursor symbol on the screen by continually sampling the analog values produced by the user's movement of the device. They include **joysticks, trackballs, touch screen panels, data tablets**, and **mice**. The latter two are the most common locator devices for editing applications. The data tablet is a flat, rectangular, electromagnetically sensitive panel. Either a ballpoint-pen-like **stylus** or a **puck**, a small (approximately 3" square x 1" high) device which fits in the palm of the hand, is moved over the tablet surface. The tablet returns to a system program the coordinates of the point on the data tablet on which the puck or stylus is currently located. The program can then map these data tablet coordinates to screen coordinates and move the cursor to the corresponding screen position. The mouse is another hand-held device about the same size as the puck. As it is moved on a flat surface, the motion of the mouse causes relative changes in x and y to be sampled by a system program. These mouse coordinates are again mapped to screen coordinates to move a cursor on the screen.

A locator device coupled with a button device allows the user to specify either a particular point on the screen at which text should be inserted or deleted, or the start and end points of a string of characters to be operated upon. In fact, the mouse and puck usually have built-in buttons for the user to signal a selection. When the cursor has been positioned over an element, the user presses a button to indicate the selection; the system correlates the cursor position with the element it covers and performs the appropriate action.

Text devices with arrow (cursor) keys are often used to simulate locator devices. These keys each show a pictured arrow, pointing up, down, left, or right, respectively. Pressing an arrow key typically generates an appropriate character which the program then translates to

update the cursor position in the direction of the arrow on the key pressed. `Up`, `down`, `left`, and `right` keys must be used sequentially to position the cursor a character at a time, aided typically by continuous movement in "repeat mode," which the device enters if a key is held down for more than, say, a second.

Still in the research stage, **voice input devices**, which translate spoken words – both literal text and commands – to their textual equivalents, may prove to be the text devices of the future. While currently restricted to a small vocabulary (typically fewer than 1000 words), voice recognizers may soon be commercially viable for command recognition.

1.2.1.2 Output Devices

Formerly limited in range, output devices for editing are diversifying. The output device serves to let the user view the elements being edited and the results of the editing operations. The first generation output devices were the (now largely obsolete) teletypewriters and other character-printing terminals, which generated output on paper. Next, "glass teletypes" based on **cathode ray tube (CRT)** technology used the CRT screen essentially to simulate a hardcopy teletypewriter, though a few operations, such as backspace, were performed more elegantly. Today's advanced CRT terminals use hardware assistance for such features as moving the cursor, inserting and deleting characters and lines, and scrolling lines and pages. The new generation of professional workstations, based on personal computers with high-resolution raster displays, support multiple proportionally-spaced character fonts to produce realistic facsimiles of hardcopy documents. Thus, the user can see the document portrayed essentially as it would look when printed on paper.

While many editors, especially traditional ones, have interfaces that allow them to run on both CRT and hardcopy terminals, we concentrate on the more sophisticated capabilities that can readily be implemented only on CRT terminals with user-manipulable cursors. Much of the discussion does, however, apply to both hardcopy and CRT output devices.

1.2.1.3 Interaction Language

The interaction language of a text editor can be divided into three parts: the **semantic component**, the **syntactic component**, and the **lexical component** [FOLE82]. The semantic component of the language specifies functionality: what operations are valid for each element, what information is needed for the manipulation of each element, what the results of the operations are, and what errors may occur. The semantic component defines meanings of the specified operations, not particular data structures or dialogues to implement those operations.

The syntactic component specifies the input and output rules by which **tokens**, as the atomic elements, are put together to form sentences in the **grammar** of the language. In terms of editor input, the set of tokens might include character strings, commands, and screen positions. In terms of output, the set of tokens might include character strings, lines, and formatted paragraphs. The syntax must be easy for the user to learn and remember, and must follow naturally from the conceptual model of the system. The syntax of interaction languages is generally one of three sorts: **prefix, postfix,** or **infix**. A prefix (verb/noun) command specifies the operation desired, followed by the element(s) which that operation will affect. A postfix (noun/verb) command specifies just the opposite: the element to be operated upon is specified first, followed by the operation. An infix (noun/verb/noun) command is a cross between the two former types: the operation is surrounded by its operands. Infix typically only is needed when there exists more than one operand. Many languages are basically postfix, but rely on infix in the cases where more than one operand is needed. In a **cross-product** interface, the user can match a noun from a list of nouns with a verb from a list of verbs to form the appropriate

command. In other types of interfaces, the verb/noun distinction is not always clear-cut; a single editor command, such as **delete-word**, can be composed of both a verb and noun. Here the user specifies the command `delete` and the element is particularized to the nearest unit of the corresponding type `word`.

The lexical component specifies the way in which **lexemes**, information from the input devices or for the output devices, are combined to form the tokens used by the syntactic component. Thus typing the lexemes `D`, `-`, `W` followed by a carriage return would result in the `delete-word` token.

The interaction language is generally one of several common types, based on the manner of specifying lexemes. The **typing-** or **text command-oriented** interface is the oldest of the major editor interfaces. Here, the user communicates with the editor by typing text strings both for command names and operands. These strings are sent to the editor and are usually echoed on the output device.

Typed specification often requires the user to remember the exact form of all commands, or at least of their abbreviations. (Some systems will prompt the user with valid choices if an ambiguous command abbreviation is typed.) If the interaction syntax is complex, the user must continually refer to a manual or an online "help" command for a description of less frequently used commands. Additionally, typing is time-consuming, especially for the beginner. The **function-key** interface addresses these deficiencies. Here each command has associated with it a marked key on the user's keyboard. For example, the `insert character` command might have associated with it a key marked `IC`. Forgetting the commands is unlikely, since they are literally at the user's fingertips. For the common functions, usually only a single key need be pressed. Function-key syntax is typically coupled with cursor-key movement for specifying operands, thereby eliminating much typing. Advocates cite "muscle memory" of key location as a prime advantage of function key interfaces.

For less frequently invoked commands or options in a many-function editor, an optional textual syntax may be used. More commonly, special keys "shift" the standard function key interpretations, just as the shift key on a typewriter shifts the standard interpretation of a key from being lowercase to being uppercase. As an alternative to shifting function keys, the alphanumeric keyboard is often **overloaded** to simulate function keys. The user again uses special shift keys – the most common being the **control**[3] key depressed simulataneously with normal keys to generate new characters that are interpreted like function keys. Generally, functions are assigned to alphanumeric keys in two ways: topologically and mnemonically. In topological layout, functionally similar keys are grouped in close proximity; for example, in Brown's *bb* [REIS81], the control key coupled with the `Q`, `W`, `E`,or `R` keys correspond to `delete word`, `delete to end of line`, `delete to blank line`, and `delete paragraph`, respectively, while the keys directly below, `A`, `S`, `D`, `F`, coupled with the control key, correspond to `insert word`, `insert to end of line`, `insert to blank line`, and `insert paragraph`. Another topological layout is seen in Wordstar [MICR81], in which the cursor commands, `up`, `down`, `left`, and `right` are bound to the `CTRL-E`, `CTRL-X`, `CTRL-A`, and `CTRL-F` keys to make use of their relative directions on the keyboard. Mnemonic layout binds commands to keys whose letters or symbols invoke some type of mental image or connection. For instance, in the EMACS [STAL80] default key bindings, `CTRL-B`, `CTRL-E`, `CTRL-F`, `CTRL-R`, and `CTRL-K` stand for `Backward character`, `End of line`, `Forward character`, `Reverse search`, and `Kill line`, respectively.

[3] In examples, the control key will be abbreviated CTRL- and the escape key ESC.

Typing-oriented systems require familiarity with the system and language, as well as some expertise in typing. Function-key-oriented systems often have either too few keys, thus requiring keyboard-overloading by binding single keys to several interpretations and necessitating multiple keystroke commands, or have too many unique keys, resulting in an unwieldy keyboard. In either case, even more agility is demanded of the user than by a standard keyboard. The **menu-oriented** user interface is an attempt to address these problems. A **menu** is a multiple-choice set of text strings or **icons** (graphical symbols which represent objects or operations) from which the user can select items to perform actions [GOLD79]. The editor prompts the user with a menu of only those actions that may be taken at the current state of the system. The user knows that if a command appears in the menu, it can be selected, and the typical "you can't do that operation in this *(x)* mode" error messages that occur with other types of interfaces are eliminated.

One problem with a menu-oriented system can arise when there are many possible actions and multiple choices required to complete an action. Since the display area for the menu is usually rather limited, the user might be presented with several consecutive menus in a hierarchy before the appropriate command and its options appear. Since this can be annoying and detrimental to the efficiency of a seasoned user, some menu-oriented systems allow the user to turn off menu control, leaving a language- or function-key-oriented editor as a base. Others have the most-used functions on a main command menu and have secondary menus to handle the less frequently used functions. Still others display the menu only when the user specifically asks for it. For instance, in more modern editors based on a Smalltalk-80*-like interface [TESL81, GOLD82] (see Figure 1), **pop-up** menus, in response to the user's button-push, instantly appear on some part of the screen near the cursor (perhaps temporarily overlapping some existing information) to give the user the full choice of applicable commands. The user selects the appropriate command with a mouse; the system executes the command and the menu disappears instantly. Interfaces like this, in which prompting and menu information are given to the user at little added cost and little degradation in response time, are becoming increasingly popular.

While the semantic component of the language is similar from system to system (at least for simple operations), the lexical and syntactic specification of commands vary widely. For example, to delete the word "`bazinga`" in an editor using typed commands from a keyboard without cursor keys, one might search for an occurrence of the pattern "`bazinga`" and then type `delete/bazinga`; in another, using typed commands with cursor keys, one might type `dw` after driving a cursor to point at any character in the word "`bazinga`"; in a function-key driven system, one might press the `del word` key (or keys) after pointing at "`bazinga`"; and in a menu-oriented system, one might point to and then `select` the '`b`' in "`bazinga`," adjust the selection to the entire word by pushing the `adjust` button, and then select the `delete` icon from the displayed menu. The interaction language can be viewed as a layered architecture: similar semantics can be reflected in a variety of syntaxes, while similar syntaxes can be reflected in a variety of lexical styles. For example, given semantic routines that perform insertion and deletion in an unbounded stream of characters, designers could provide a prefix or postfix syntax for specifying `insert` or `delete` commands. Similarly, given either the prefix or postfix syntax, designers could provide typing-oriented, function-key-oriented, or menu-oriented lexical input styles. This layering invites device-independence, since differing lexical forms can be mapped into the same syntax. The layering also allows syntax-independence, since differing syntactical styles can be mapped into the same semantic operations.

---

* Smalltalk-80 is a registered trademark of Xerox Corporation.

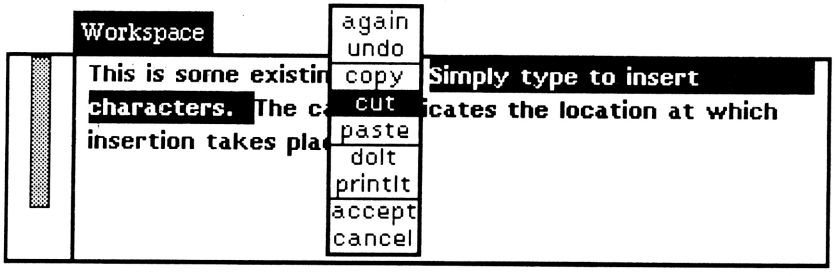


Figure 1. *A Smalltalk-80 Pop-up menu.*
Here the arrow cursor, driven by the mouse, points to the **cut** item in the pop-up menu temporarily overlapping the text window. When the proper mouse button is pressed, the command is executed and the menu, no longer needed, disappears.

1.3 The Editor: A System Viewpoint

Editors in general follow an architecture similar to that in Figure 2, regardless of the particular computers on which they are implemented and the features they offer.

The **command language processor** accepts input from the user's input devices, lexically analyzes and tokenizes the input stream, syntactically analyzes the accumulated stream of tokens, and, upon finding a legal composition of tokens, invokes the appropriate semantic routines.

At the syntactic level, the command language processor, like a programming language processor, may generate an intermediate representation of the desired editing operations instead of explicitly invoking the semantic routines. This intermediate representation is decoded by an interpreter that invokes the proper semantic routines. This allows the use of multiple interaction syntaxes with a single set of semantic routines that are driven from a common intermediate representation.

The semantic routines invoke traveling, editing, viewing, and display. While editing operations are always specified explicitly by the user, and the display operations implicitly by the other three categories of operations, traveling and viewing operations may be either explicitly specified or implicitly invoked by the editing operations. In fact, the relationship between these classes of operations may be considerably more complicated than the simple model of Section 1.1 (travel to determine *where* the selection should take place, filter to select *what* is to be viewed and manipulated, format to determine *how* the view is to appear, then edit and reformat) might

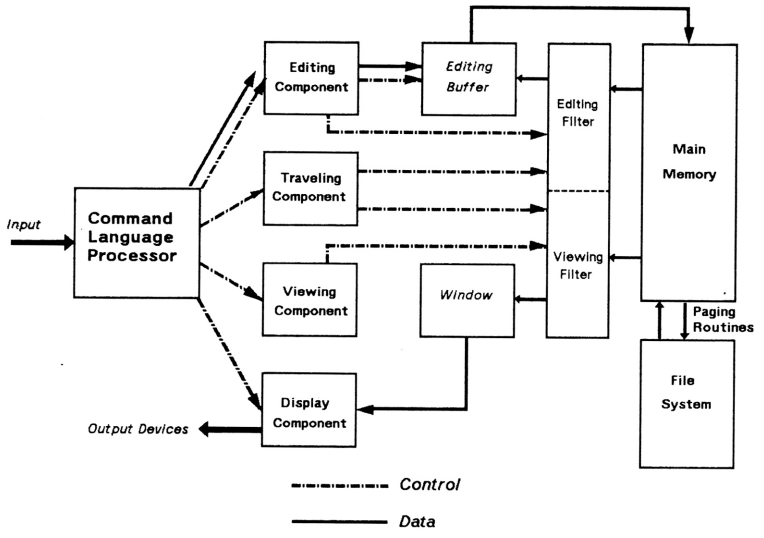


Figure 2. *The Editor – A System Architecture*

suggest. In particular, there need not be a simple one-to-one relationship between what is "in view," i.e., what is displayed on the screen currently, and what can be edited. To illustrate this, we take a closer look at the *conceptual* components of Figure 2 below.

In editing a document, the start of the area to be edited is determined by the **current editing pointer** maintained by the **editing component** of the editor, the collection of modules dealing with editing tasks. The current editing pointer can be set or reset explicitly by the user with a traveling command such as `next paragraph` and `next screen`, or implicitly by the system as a side effect of the previous editing operation, such as `delete paragraph`. (The **traveling component** of the editor actually performs the setting of the current editing and viewing pointers, and thus the point at which the viewing and/or editing filtering begins.) When the user issues an editing command, the editing component invokes the **editing filter**. The editing filter filters the document to generate a new **editing buffer** based on the current editing pointer as well as the editing filter parameters. These parameters are specified both by the user and the system, and provide such information as the range of text that can be affected by the operation, for example the "current line" in a line editor or the "current screen" in a display editor. Filtering in both

the case of editing and viewing may be defaulted to the selection of contiguous characters at the current point or depend on more complex user specifications pertaining to content and structure of the document to gather not-necessarily contiguous portions of the document, as discussed in Section 3.1. The semantic routines of the editing component then operate on the editing buffer, which is essentially a filtered subset of the document data structure. (Note that this explanation is at the conceptual level – in a given editor, filtering and editing may be interleaved, and no explicit editing buffer created.)

Similarly, in viewing a document, the start of the area to be viewed is determined by the **current viewing pointer** maintained by the **viewing component** of the editor, the collection of modules responsible for determining the next view. The current viewing pointer can be set or reset explicitly by the user with a traveling command or implicitly by the system as a side effect of the previous editing operation. When the display needs to be updated, the viewing component invokes the **viewing filter**. The viewing filter filters the document to generate a new **viewing buffer** based on the current viewing pointer as well as the viewing filter parameters. These parameters, again, are specified both by the user and the system, and provide such information as the number of characters needed to fill the display and how to select them from the document. The viewing buffer may contain the "current line" or the null string (for "silent" feedback) in line editors, while in screen editors it may contain a rectangular cutout of the quarter plane of text. This viewing buffer, is then passed to the **display component** of the editor, which maps it to a **window** or **viewport**, a rectangular subset of the screen, to produce a display.[5]

The editing and viewing buffers, while independent, can be related in many ways. In the simple case they are *identical*, as in the case of screen editors, in which the user edits the material directly in view on the screen, rather than specifying material with typed commands (see Figure 3).

The editing and viewing buffers can also be *disjoint*. For example, in the Berkeley UNIX* editor *ex* [JOY80a], a user might travel to line 75, and after viewing it, decide to change all occurrences of "ugly duckling" to "swan" in lines 1 through 50 of the file by using the substitute command:

```
1,50s/ugly duckling/swan/
```

As part of this editing command, there is implicit travel to the first line of the file, lines 1 through 50 are filtered from the document to become the editing buffer, and successive substitutions take place in the editing buffer without corresponding updates of the view. If the pattern is found, the current pointers are moved to the last line on which it was found, and that line becomes the default contents of both the editing and viewing buffers, while if the pattern is not found, line 75 remains as the default editing and viewing buffers.

The editing and viewing buffers can be *partially overlapping* on a screen when the user specifies a search to the end-of-document starting at a character position in the middle of the screen. Here the editing filter creates an editing buffer that contains the document from the

[5]The term "viewport" is standard for "visible representation on the screen" in computer graphics terminology, while in editing terminology, the term "window" is used loosely to mean both the viewing buffer and the mapped representation of the viewing buffer on the screen. Unfortunately the term "window" in graphics terminology is analogous to our "viewing buffer," lending more confusion to the definitions. We have tried to limit the use of the term "window" to reduce confusion, but when we do, we use it in the editing sense of the "mapped representation of the viewing buffer on the screen." We use the term "view" to mean a "display of a filtered subset of a document in a window."

* UNIX is a trademark of Bell Laboratories.

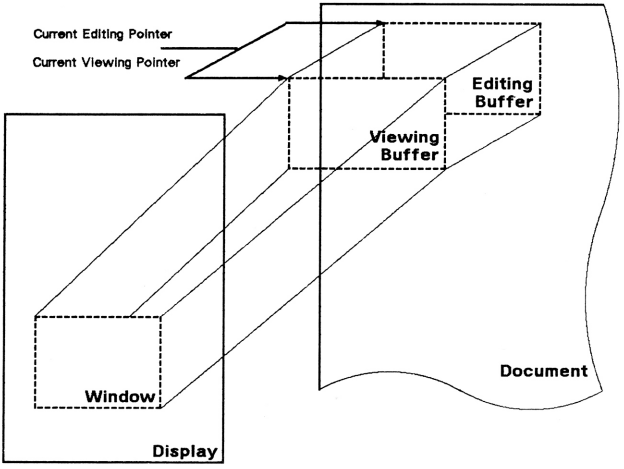


Figure 3. *Elements of the Editing Component*
The diagram above shows the relationship of the current operating pointer in a document file to the editing buffer and the viewing buffer. In this diagram, the file is indicated as a quarter plane of text. The current editing pointer points to the start of the editing buffer. The current viewing pointer points to the start of the viewing buffer. The two point to the same place, since the viewing buffer coincides with the editing buffer in this example.

selected character to the end of the document, while the viewing buffer contains the part of the document that is visible on the screen (only the last part of which will be in the editing buffer).

Finally, the editing and viewing buffers may be *properly contained* in one another. As an example of the editing buffer contained in a larger viewing buffer, the CPT word processing editor [SEYP79] allows full 8-1/2-x-11-inch page views. The user must scroll a page to put a line to be edited at the **typing bar**, analogous to rolling the platen of a typewriter to align the desired line with the typing element. Here the editing buffer corresponds to the one-line subset of the document which is in the middle of the viewing buffer. Conversely, a number of traditional editors, limited by low-speed, line-oriented output devices, provide a large editing buffer. The viewing buffer is a a small initial subset of the editing buffer to allow quick regeneration of the view; the entire scope of the editing operation is need not be in view. The FRESS editor [VAND71a, PRUS79] allows the user to select the number of lines and the number of characters per line for the viewing buffer (the default being a single line the width of the display device for alphanumeric terminals) and also allows the user to select the size of the editing buffer (the default being 2000 characters); the viewing filter selects as many characters as necessary, starting at the beginning of the editing buffer, to provide the viewing buffer for the necessary display.

Windows typically cover either the entire screen or a rectangular portion of it. Mapping to windows that cover only part of the screen is especially useful for editors on modern, high-resolution raster-graphics-based workstations, as they allow the user to view and to interact with multiple views, allowing inter- and intra-file editing and "cutting and pasting." The notion of multiple viewing buffers and multiple windows showing differing portions of the same or multiple files at the same time on the screen is designed into such editors. Alternatively, on graphics-based workstations with multi-window **display managers** or **window managers** that allow the user to manipulate the placement of windows on a screen [LRG76, TEIW77, LANT79, MEYR81, SYMB81, TESL81], the user can run a one-viewing-buffer editor in each of the many windows. It is through the support of the display manager, *not* of the editor, that the user can cut and paste text between multiple windows containing portions of the same file or portions of multiple files.

As a comment on slow progress in the editing field, we note that in the database and graphics fields, the notion of multiple simultaneous views of the same or related data is well understood and commonly accepted. Though the facility for multiple views of documents at varying levels of detail and appearance has been available in the NLS editor since the mid-1960's [ENGE68] (see Section 2 and Part II, Section 4.5), this powerful organizational mechanism is only now beginning to gain acceptance in the editing community.

The viewing buffer-to-window mapping is accomplished by two components of the system. First, the viewing component formats an ideal view, often expressed in a device-independent intermediate representation. This ideal view can range from simply a window's worth of text, arranged so that lines are not broken in the middle of words ("ragged right" justification), to a facsimile of a page of fully formatted/typeset text with equations, tables, and figures.

Second, the **display component** takes this idealized view from the viewing component and simply maps it to a physical output device in the most efficient manner possible. If a view that has been computed by the viewing component cannot be fully displayed because the window is partly obscured by another window, this mapping becomes somewhat more complicated.

Updating of a full-screen display connected over low-speed lines (1200 baud or less) is slow if every modification requires a full rewrite of the display surface. Much research is concerned with optimal screen-updating algorithms that compare the current version of the screen with the following version and, using the innate capabilities of the terminal, write only those characters needed to generate a correct display. Typical algorithms for this intelligent screen update are described in ARNO80, BARA81, GOSL81, and STAL81.

Device-independent output, like device-independent input, promotes interaction language-portability. This decoupling of editing and viewing operations from display functions for output is important for purposes of portability: it is unwieldy to have a different version of the editor for every particular output device. Many editors make use of a **terminal control database** [JOY81]. Instead of having explicit terminal-control sequences in display routines, these editors simply call terminal-independent library routines, such as `scroll down` or `read cursor position`, that look up the appropriate control sequences for the host terminal. Consequently, adding a new terminal merely entails adding a database description of that terminal.

In addition to the interrelated traveling, viewing, editing and displaying components, special **utility components** may exist to provide a variety of services to aid the user in document production such as spelling checkers/correctors and word-count analyzers.

The components "communicate" with a user document on two levels: in main memory and in the disk file system. At the beginning of an editing session the editor first asks the file system

to open the appropriate file and then reads the entire file or files, or portions thereof, into main memory. Loading an entire document into main memory may be infeasible. Yet if only a portion of the document were resident in main memory and if many user-specified operations entailed a disk-read by the editor to load the affected portion, editing would be unacceptably slow. In many modern systems this problem is solved by mapping the entire file into virtual memory and letting the operating system perform efficient demand-paging. Alternatively, in systems without virtual memory or with limited virtual memory per user, **editor paging routines** are required. These read in one or more logical chunks of a document, called pages (though there is typically no correspondence between these pages and hardcopy document pages or virtual memory pages), into main memory, where they reside until a user operation requires another piece of the document. In either case, documents are often represented not as sequential strings of characters, but as an **editor data structure** that allows addition, deletion, and modification with a minimum of I/O and character movement. When stored on disk, the file may be stored in terms of this data structure or in an editor-independent, general-purpose format (e.g., as character strings with imbedded control characters such as linefeed and tab).

1.3.1 Configurations

Editors function in the three basic types of computing environments: **timesharing**, **standalone**, and **distributed**. Each type of environment imposes some constraints on the design of an editor. The timesharing editor must function swiftly within the context of the load on the computer's processor, primary memory, and secondary memory. The editor on a standalone system must have access to the functions that the timesharing editor obtains from its host operating system; these may be provided in part by a small local operating system or may be built into the editor itself if the standalone system is dedicated to editing. The editor operating in a distributed resource-sharing local network must, like a standalone editor, run independently on each user's machine, and must, like a timesharing editor, contend for shared resources.

Some timesharing-based editing systems take advantage of local (terminal-based) hardware to perform editing tasks. These **intelligent terminals** have their own microprocessors and local buffer memories in which editing manipulations can be done, thus saving the time needed to read and write main computer memory, but adding tricky data structure synchronization problems. Small actions are not controlled by the CPU of the host processor but are handled by the local terminal itself. In a system using an IBM 3270 series terminal, for example, the editor sends a full screen of material from the mainframe computer to the terminal. The user is free to add and delete characters and lines; when the buffer has been edited, its updated contents are transmitted back to the mainframe. The advantage of this scheme, that the host need not be concerned with each minor change or keystroke, is also the major disadvantage. With a non-intelligent terminal the CPU "sees" every character as it is typed in and can react immediately to perform error checking, to prompt, to update the data structure and to record or "journal" the keystrokes for undoing editing operations (see Section 3.4). With an intelligent terminal, the lack of constant CPU intervention often means that the functionality provided to the user is more limited. Also, local work on the intelligent terminal is lost in the event of a system crash. Conversely, systems which allow each character to interrupt the CPU may not use the full hardware editing capabilities of the terminal because the CPU needs to see every keystroke and provide character-by-character feedback.

As the local-area network of largely self-sufficient workstations becomes the dominant architecture, we can expect that the problem of synchronizing intelligent terminals with timeshared hosts will disappear and that character-by-character feedback will become the norm.

2. HISTORICAL DEVELOPMENT OF EDITORS

The history of editing is one of many complementary development efforts proceeding in parallel. Editors are so numerous and their relationships so cloudy, that it is very difficult to provide an accurate chronology. Rather, we briefly overview some of the important concepts, citing familiar and representative examples. (For a survey of specific editors through 1971, see VAND71b.)

Non-interactive computerized editing began with the manipulation of "unit record" punched cards. The basic unit of information was the 80-column line; the user made corrections on a line-by-line basis, retyping mistyped cards. Compared to toggling in bits at the system console, the card gave the programmer new freedom. One could store information in both human- and machine-readable form and one could "travel" through this information, changing its order, discovering and fixing errors, recognizing color-coded groups of cards, or simply browsing through the deck.

Punched card decks had many disadvantages, such as the "rearrangement" that resulted when a box was accidentally dropped. More seriously, editing a small part of a large document required feeding the entire document, often thousands of cards, into the reader for every change. To correct small errors such as single character errors or double-character transpositions, the user had to retype the offending characters and to replicate the other characters with the duplication facilities of the keypunch. Replacing a word with a word of a different size necessitated duplicating all characters prior to the word and retyping all the characters from the new word forward. If the incorrect card was almost completely filled with characters, inserting a new word might result overflow of the contents, causing the insertion of one or more new cards to handle the overflow. Performing a global change required manually finding every occurrence of the old pattern and again manually replacing it with the new pattern; if the new pattern were larger than the old pattern, multiple overflows could easily occur.

To address the problems of the punched card in the still predominantly batch environments of the 1960's, **card** or **batch editors** were created. Here the programmer's initial deck of cards was stored as a card-image tape or disk file. Each card was referenced by a unique sequence number. Changes were made by creating an **edit deck** composed of cards containing editing requests, and running the deck through the batch editor program. For example, the request "in card 35, correctly spell the word 'rate'," would be made by simply typing the desired sequence number **35** on one card followed by a card containing the new contents of line 35, or more simply, by composing a card with a sequence number and an editing command, as in

```
35 CHANGE/RATA/RATE/
```

Batch editors removed the problems of dropped cards and of retyping (in many cases), and, in some versions, provided new operations such as global replacement of a pattern. There were several disadvantages, however. Programmers needed to have a lineprinter listing of the entire card deck before making any change. Also, some of the organizational characteristics offered by cards, such as the easy visual inspection of a properly sequenced, color-coded, and well-labeled card box, were lost.

Systems like IBM's MTST [IBM67], which used a Selectric typewriter as an input device and small magnetic tapes and/or cards as storage media, were the forerunners of today's word processing systems. Because these primitive interactive editors relied on sequential storage media such as magnetic tape or magnetic cards, the user could only step through card images linearly, stopping at lines which needed correction and retyping them. To go backward, the user had to "rewind" and then "fast-forward" the file to the desired place. In addition, the utility of these

initial line editors was limited by the typewriters, which supported the viewing of only one line at a time and had very slow printing speeds.

With the advent of timesharing in the mid-1960s, interactive **line editors** were designed that allowed the user to create and modify disk files from terminals. These editors attached either fixed or varying ("varying" meaning "sequential relative to the top of the file") line numbers to lines of limited length (initially 80 characters), allowing the user to reference a unit of information. Early examples of these include ATS [IBM70] and VIPcom [VIP69]. Simple command languages allowed the user to make corrections within a line or even within a group of contiguous lines, using much the same syntax as batch editors. Some early timesharing editors still restricted the user to forward access; later, this restriction was lifted and the user could scroll both forward and backward through a file. Typically, line editors with bounded-length lines shared the unfortunate property of **truncation**: if an insert or change of characters would force the line to exceed the maximum length, characters were dropped off the end of the line as needed. This implementation "feature," stemming from a conceptual model of the editing process based on simulating punched cards and line printer listings, was only marginally acceptable for program editing and unacceptable for serious manuscript preparation. (In the latter case, automatic creation of a new line upon overflow would have been a trivial fix.)

Another advance was the creation of the **context-driven line editor**, which allowed the user to identify the line containing the target of an operation by specifying a character context pattern for the editor to match, rather than by giving an explicit line number. One of the first examples of the context-driven line editor was the trendsetting editor running on the IBM 7090 as part of Project MAC's CTSS [CRIS65]. Other classic examples include IBM's CMS editor [IBM69] (see Part II, Section 4) and Stanford's WYLBUR [FAJM73]. At this point in the history of editing, users were still forced to think about multi-line entities, such as paragraphs and program blocks, as groups of integral lines, usually in card-image format; no interline commands were available that would, for example, delete text spanning from the middle of one line to the middle of the next line.

The first break from the 80-column card image came in the form of **variable-length line editors**, typified by Com-Share's Quick Editor (QED) [DEUT67, COMS67]. The main element of operation was still the line, but now each line could be of "arbitrary" length. Initially, these lines were actually limited to some maximum. QED popularized the notion of a "superline" (limited to 500 characters in length), which the online display process broke into viewable lines of 80 characters each until the superline was exhausted. While editing across superline boundaries was still impossible in these editors, the probability that a full phrase or sentence that needed editing would fall within a single superline was much greater than with an 80-character limit. Later, true variable-length line editors removed the restriction. By removing the card-image orientation of the editor, the variable-length line editor had strong and beneficial impact on the versatility of text processing. Another far-reaching result of the invention of variable-length line editors was that displayed text was no longer considered to be a one-to-one mapping of the internal representation, but rather a tailored, more abstract view of the editable elements.

Even with superline editors, three basic problems in manuscript editing remained: truncation when the line length was exceeded, inability to edit a string crossing line boundaries, and inability to search for a pattern crossing line boundaries. This last problem is an especially pesky one when transcribing editing changes from formatted hardcopy in that even a short phrase which appears on one line within the paper may be spread across two lines in the document's source file, unbeknownst to the user. Consider, for example, the familiar "the the" typo

problem. If a document being edited on a line editor contained the lines:

```
...The power of the
the stream editor...
```

a search command:

```
locate/the the
```

would not find the pattern "`the the`," since it appears on two separate lines. The **stream editor** concept solved all three problems by eliminating line boundaries altogether: the entire text was considered a single stream or string that was broken into screen lines by display routines. An arbitrary string between any two characters could be defined for searching and editing. HES [CARM69], and FRESS are examples of stream editors. Although the TECO stream editor left the carriage returns and line feeds in the text stream, these could be edited like any other character; they did not serve to isolate one line from the next in editing. In fact, the generality of the TECO stream model has made possible the construction of several sophisticated editors using TECO as a nucleus (see Part II, Section 4).

Another way of dealing with the limitations of line/superline editors was to use the power of multi-line display screens, which provided cursor addressability and (possibly) local buffers, to create what are called synonomously **full-screen, display,** or **cursor editors**. These editors work either with variable-length lines or with streams, offering the user an entire screenful of text to view and edit without regard to line boundaries. An early example of a time-shared display editor is Stanford University's TVEDIT [TOLL65, McCA67]. Commands, represented by control character sequences, could be interspersed with the input of "normal" text. Users were able to move the cursor to point to the text they wished to manipulate rather than having to describe text arguments in some awkward syntax. Characters could be replaced by simply typing over them. Characters could be deleted by placing the cursor on the character and pressing the delete control character; characters to the right of the cursor moved left so that the cursor seemed to "swallow" characters. Similarly, for insertions, the characters to the right of the pointer moved to the right, "making room" for the new characters. The TVEDIT concepts and similar work by Ned Irons [IRON72] form the basis of many screen editors in use today.

A major new way of thinking about editing was introduced as early as 1959 by Douglas Engelbart at Stanford Research Institute. His NLS (oNLine System), implemented in the 1960s to create an environment for online thinking and authoring, showed the power of display terminals, multi-context viewing, flexible file-viewing, and a consistent user interface [ENGE63, ENGE68, ENGE73]. One of the NLS project's many important contributions was the mouse, an input device which is only now being integrated into commercial products. NLS was the first structure editor in that it provided support for text structure and hierarchy, not just for manipulating raw strings of text: the user could manipulate documents in terms of their structure, not only their content. NLS and other related systems, such as HES, FRESS, and Xanadu* [NELS74, NELS81], were particularly important because they view the editor as an *author's tool*, an interactive means for organizing and browsing through information, rather than simply as a mundane tool for altering characters in a single file.

Hansen's EMILY [HANS71] extended the concept of the structure editor and developed the **syntax-directed editor**, in which the structure imposed on a program being edited was the structure of the programming language itself. Users were able to manipulate logical constructs, such as `do-while` loops and their nested contents, as single units.

* Xanadu is a registered trademark.

In the late 1960's, general-purpose timesharing facilities typically supported only simple interactive line-editing and batch-formatting facilities for lineprinter output. These "value-added" facilities were barely adequate enough to create and modify programs and rudimentary documentation. By the early 1970's, text processing had become sufficiently important that it became the single dedicated application on both standalone and timeshared ("shared-logic") minicomputers. Since these minis did not need to support general-purpose computing facilities, manufacturers were able to offer comprehensive editing and formatting/typesetting capabilities as well as features oriented towards document production such as database management, information retrieval, workflow management, and print- and job-queue management that were usually unavailable on general-purpose systems. For a time, owners of these systems often had more text-processing power than those with much more expensive and much larger general-purpose computers. Examples of dedicated word processing systems include CPT, Lanier, QUIP, DEC Word/11, NBI, and Wang.

An important milestone in text editing and text processing was the early-70's development and mid-70's acceptance of the UNIX* timesharing system, the first general-purpose computing environment in which text utilities were given as much weight as programming utilities. In UNIX, a suite of utilities (the *ed* text editor, the *troff* text formatter, the *eqn* equation formatter, the *tbl* table formatter, the *refer* bibliographic database and formatter, the *spell* spelling corrector, and the *style* and *diction* text analyzers [KERN82]) introduced and popularized an extensive set of text tools in the general-purpose computing community. At the same time the publishing industries – newspapers, magazines, wire services, the graphics arts – converted wholesale to electronic typesetting and layout of pages, borrowing ideas from traditional computer-based text processing, and also channeling ideas in typesetting and page-layout back to the computing community.

In the then-separate area of computer graphics, picture editors were being designed to allow the user to manipulate graphical elements. Interactive drawing techniques from Sutherland's pioneering Sketchpad system [SUTH63] were later incorporated into editor interfaces. The Carnegie-Mellon tablet editor [COLE69] is an example of the use of this technology. In this experimental editor, hand-drawn proofreader's symbols were used to edit displayed text. The symbols were drawn on a data tablet and were recognized by the program by passing various characteristics for a given symbol through a decision tree. For a **delete** or **substitute** operation, for instance, the user drew a line through the text to be deleted. The system deleted this line, blinked the indicated text for verification, separated the text by opening a blank line, and inserted a cursor, enabling new text to be typed in from the keyboard. For a **transpose** operation, the user simply used the familiar transposition mark. With the existence of data tablets with built-in microprocessors dedicated to this recognition task [IMAG81], we may expect to see this technique used in commercial editing products in the near future.

The major innovations of the 1970's in text handling and the user interface, specifically the Bravo editor [LAMP78] and the Smalltalk environment, took place at Xerox's Palo Alto Research Center (Xerox PARC). These systems demonstrated the expressive power of blending text and graphics on a high-resolution, bitmapped, raster graphics screen, using a dynamic graphical interface provided by a dedicated personal computer. Editing was done by selecting items on the screen, using the mouse as a pointing device. These systems were also the first **interactive editor/formatters**, in which the user's text was displayed on the bitmapped screen in a facsimile of the typography and layout of the final document as the document was being input

*UNIX is a trademark of Bell Laboratories.

or modified. For the first time, the user was given a notion of not only the up-to-date content, but also of the up-to-date form of the document.

Current research in the field of editing is focused upon several overlapping areas. One is that of providing a consistent, editor-based interface throughout a computer system [GOLD79, LANT80, FRAS80, APOL81, STAL81, TESL81, GOLD82]. This allows many common functions, such as renaming files, searching through directories, and debugging programs, to be performed as editing operations. For instance, to rename a file, one would type over the old filename in a listing of available files that would appear on the screen; in debugging a program, one would be able to edit the values of displayed variables. Other research topics include generalized structure editors, powerful syntax-directed editors with program-tracing capability, and interactive editor/formatters. Research in these and other areas is discussed more fully in Part II.

3. FUNCTIONAL CAPABILITIES

In this section, we take a more detailed look at interactive editors, examining their functional capabilities from a user perspective.

3.1 Traveling

User-specified traveling commands cause the **current editing and viewing pointers** of the editing and viewing components to be moved, resetting the filters that extract the editing and viewing buffers.[9] Traveling ranges from simple motion such as moving to a subsequent screenful of data, to more complicated movement such as pattern search and tree or directed-graph traversal.

Early systems, and even many of today's word processing systems, are oriented toward transcription of editing changes from hardcopy. They therefore provide relatively unsophisticated traveling mechanisms. For online authoring, however, a system should be oriented to browsing, studying, and organizing as well as to composition and revision. In such a system, traveling flexibility and power are mandatory, as the NLS experience has shown [ENGE68].

3.1.1 Simple movement

Editors commonly support two types of **intrafile motion**: **absolute** and **relative**. An absolute specification indicates a destination independent of the current position, while a relative specification indicates the destination relative to the current position. Generally, the current pointer is maintained internally to point to the corresponding current position in the data structure; in editors with pointing devices, the cursor usually points at this position in the visible text with the current pointer in lockstep. The user can move the editing buffer and viewing buffer from this position by issuing simple commands. In IBM's line-oriented CMS editor, moving the editing buffer and the viewing buffer five lines ahead in the document would be accomplished by typing:

```
next 5
```

In line-oriented editors, line numbers can be **fixed** or **varying**. In editors with fixed line numbers, numeric labels provided by the user or the system, the user can easily specify an absolute `goto` to that number. As an example, the editor portions of BASIC interpreters use fixed line numbers specified some interval apart, such as:

```
00010 FOR I = 1 TO 10
00020     J = I * I
00030 NEXT I
```

To add a print statement after the increment statement, one could simply type

```
00025     PRINT I,J
```

and the editor would put the new line between lines 20 and 30, as indicated by its numeric label.

In contrast, editors with varying line numbers keep track of a line's position internally as an offset from the top of the file. When a new line is added, the internal line numbers of all lines beneath the new line are incremented by one. Since the line numbers change dynamically, the deletion (insertion) of a line near the top of the file causes all the line numbers below it to be decremented (incremented) by one. Because of this, editing by specification of line number is best done from the bottom of the file upwards, since insertions or deletions will typically only

[9] To simplify the remainder of our discussions, we will group both pointers together under the term "current pointer."

affect the lines at and below the current pointer. This is a restriction palatable (if at all) only for transcription from hardcopy, not for online revisions.

As we have seen, the user of a display editor travels by pushing function keys such as

```
+LINE or -LINE
+WORD or -WORD
+PAGE or -PAGE
+1/2 PAGE or -1/2 PAGE
```

that effectively change the contents of both the editing and viewing buffers. Additionally, the editing and viewing buffers can be moved left or right by a specified number of character positions (columns) to allow editing of wide lines, as in the following command:

```
50 LEFT
```

`Top and bottom (of file)` are two very common, highly functional traveling commands.

Many editors have **mark** or **saved-position stacks** or **rings** that record locations in the file. Some maintain an implicit mark ring, automatically storing the location each time the user makes a selection for a command, and allowing the user to retrace his/her steps. Others have explicit mark buffers; the user uses the `save position` command to store the current location in any number of buffers. Additions or deletions in the file will throw off the exact location if the editor only saves a character- or line-pointer or other varying index, but will typically leave the user in a location close to the original marked location.

Systems like NLS and FRESS allow character string **labels** to be inserted in the text; these move as the adjacent text is moved. The user need not know of the location of a piece of text, whether relative or absolute, in this system, but, for example, can travel instantly to the text labeled "`casestudy`" with the command `goto-label casestudy`. In the case of program editors, identifiers in procedure declaration statements can be used as implicit labels, as in the EMACS **tags** facility explained in Part II, Section 4.

Of course, many target-dependent traveling operations exist. In document editors, for example, simple movements are provided to bring a user to the beginning of a section or chapter. In a program editor, a traveling command might take the user to the next syntactic element. More specific examples are provided in Part II, Section 4.

3.1.2 Pattern searching

The above methods of traveling are *position*-dependent: `goto line 7`, `go-back 15 lines`, or `goto-label "sec5"`. Almost all modern text editing systems additionally allow a *content*-dependent specification of location: **pattern searching**. A pattern searching command (usually called `find` or `locate`) generally changes the editing and viewing buffers so that they contain the pattern that was being sought.

Since typing an entire pattern is a tiresome operation, specification aids have been developed. The familiar "..." (**ellipsis**) construct can be specified to abbreviate a long pattern by indicating simply a few characters of context at the beginning and end of the pattern. Thus, one can locate the text string:

```
Now is the date for some good men
```

with the command:

```
locate /Now...men
```

Regular expression context patterns extend the ellipsis concept to more powerful pattern-matching capabilities, for instance, matching a pattern that does or does not contain a particular

set of characters, matching a pattern only if it occurs at the beginning of the line, or matching a pattern regardless of the case of the characters (known as **case-insensitive pattern matching**).

3.1.3 Interfile Motion

The ability to travel between two or more files is extremely useful in online authoring. At the least, the user must be provided with commands to switch between two files. (This is much more useful, of course, if the two files can be displayed in two windows on the screen.) More advanced editors maintain a circular list of the files visited; the user can travel from one to the next or the previous in an ordered fashion, or in some cases choose any of the files from a display of the circular list.

Interfile travel generates several new problems. An editor may need a more intricate internal data structure and associated editing and viewing buffers for each open file to allow multiple files to be active at once. If the editor has the ability to map file viewing buffers to multiple windows, but can only keep track of one file at a time, a file must be saved each time the user switches windows; this discourages online browsing. As well, attributes might be kept for each file so that when a user is editing a LISP program the editor matches parentheses and automatically indents lines. When the user returns to a textual document, the editor would no longer do so.

3.1.4 Hypertext and Trails

The above types of travel are usually employed by the user *during* the editing of a document, typically to get to the next place in the file where an edit is to be performed; the paths of travel are not stored from one editing session to the next. Occasions do arise when the user wants to set up (semi-) permanent paths or links within a document or between documents. The motivation for such text links is well-stated in a seminal article by Vannevar Bush [BUSH45], who envisioned the "memex" device for authors and readers in 1945:

> The human mind. . .operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain. It has other characteristics, of course; trails that are not frequently followed are prone to fade, items are not fully permanent, memory is transitory. Yet the speed of action, the intricacy of trails, the detail of mental pictures, is awe-inspiring beyond all else in nature. . . . Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, "memex" will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. . . .when numerous items have been thus joined together to form a trail, they can be reviewed in turn, rapidly or slowly, by deflecting a lever like that used for turning the pages of a book. . . .It is exactly as though the physical items had been gathered together from widely separated sources and bound together to form a new book. It is more than this, for any item can be joined into numerous trails.

Hypertext editing systems are the modern-day incarnation of Bush's pre-computer memex. Hypertext, a term coined by Ted Nelson [NELS67], is "the combination of natural language text with the computer's capacities for interactive branching, or dynamic display...a nonlinear text...which cannot be printed conveniently...on a conventional page." Simply, hypertext is defined as non-sequential writing. Hypertext systems allow the user to construct arbitrary links from any chosen point in a document to any other point in that document or in any other document in the user's domain. Menus of such links can be set up to provide a branching text that constantly evolves to provide a "dynamic and plastic structure" [ENGE68]. A batch

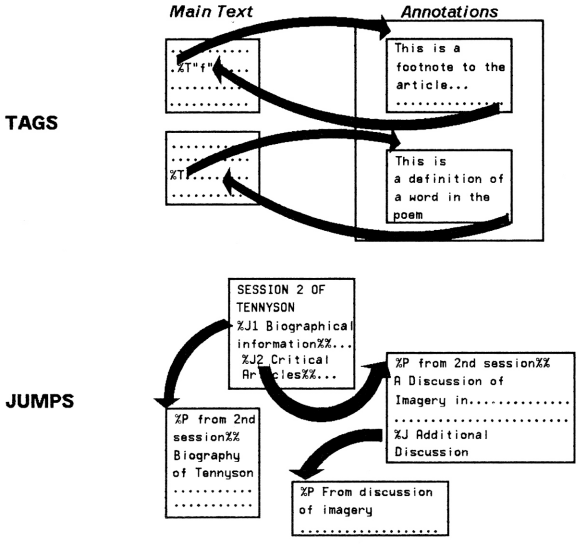


Figure 4. *An example of hypertext in FRESS.*

Tags and **jumps** are two of FRESS's hypertext elements. A tag `T` points to a single element (such as a footnote) in an "annotation space"; the user views the annotation but remains in the main text. A jump `J` indicates a path to another part of the current document or to another document. By taking a jump, one travels, changing the editing and viewing buffers of the document. In the new context, one can edit, take another jump, or take the reverse of a jump (labeled `P` for **pmuj**, jump spelled backwards), which returns to the unique source of the `J` that brought the user to the `P` in the first place. Multiple jumps to the same text would have multiple `P` labels so the user could distinguish the sources. Other system commands allow the user to retrace steps on a session basis, i.e. to return from all previous jumps with explicit `return` commands.

formatter can follow keyword-specified links to compile the text needed to produce a document. An interactive user can select links by keyword-specification or by menu-selection to lead to desired text, typically displayed in a separate window. In NLS, links can also be accessed via complex specifications of structure and/or content. One can specify, for example, "follow the first link containing the string '`edit`' in any third-level statement in the hierarchy, starting the search at the current first-level statement in view."

In the simplest sense, a hypertext document can consist of a "table of contents" with online links to files containing each chapter, each of which can link to sections. More advanced forms of hypertext include the use of links, rather than the online equivalent of standard footnotes, to point to the actual material referenced, allowing instantaneous access to cited material. Thus a document can be browsed online and predesigned but optional diversions can be set up along trails that readers may find interesting. Of course, authors and readers are free to create links to new files or to **annotation spaces** in which they may comment on what they are reading, thereby

enriching the trails for others. A hypertext system may allow the user not only to see links to other places *from* the current position, but also to see that there are links from other places *to* the current position [VAND71a] (see Figure 4). The browser is then free to jump backward to see from what document this link was issued. A hypertext document is a bidirectional network of associative trails, a constantly evolving directed graph structure of text nodes that readers and authors can traverse. Catano [CATA79], Robertson et al. [ROBG79], and Herot et al. [HERO80] provide examples of the use of this arbitrary directed-graph structure for documents. Feiner et al. [FEIN82] use the hypertext concept in the context of graphics-oriented "electronic books" (see Figure 5).

3.2 Viewing

To provide a basis for editing and browsing, and to provide feedback after an editing command is executed, the viewing component creates an up-to-date view on the display. The viewing operations are of three sorts: 1) filtering operations in which appropriate portions of the raw data structure of the file are selected for the viewing buffer; 2) operations that format this

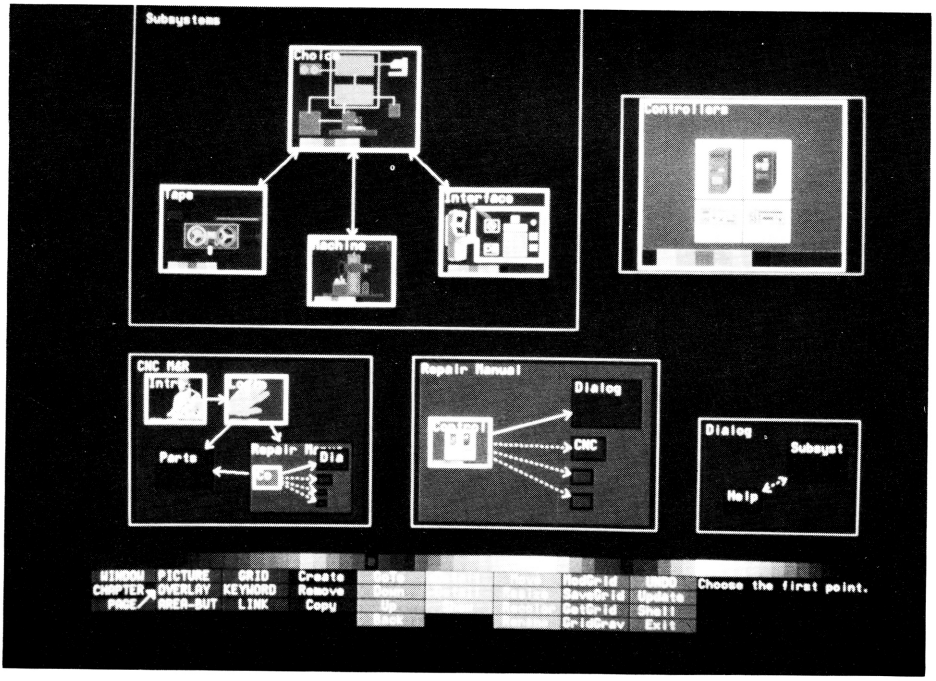


Figure 5. *Electronic Book Layout [FEIN82].*

A multiple-window display in the document layout system. The author may traverse the document's structure independently in each window and make links between chapters and pages in different windows. The author may also create, resize, and remove windows as needed, and revise the level of detail of the pages and chapters shown to control the amount of information presented.

ıltered data to produce an ideal view; and 3) operations that map this ideal view to a window ır page on a physical output-device.

3.2.1 Filtering

Editors that are especially oriented towards browsing (for example, NLS and FRESS), ısually provide facilities to alter the **viewing specifications (viewspecs)**, allowing the user to :ontrol the filtering operations. Viewspecs can be used to control any number of selection and viewing parameters (parameters controlling what is to be displayed and how, respectively) to effect **information hiding** and online formatting. NLS, for instance, allows the user to indicate what levels of detail in a hierarchy are to be displayed, as well as how many characters per statement and even which statements, on the basis of content specification (for example, "fill the window with the first 50 characters of as many of the statements that lie between section 1.3 and 1.9 as possible, up to and including the third decimal level"). Additionally, the viewspecs control whether the selected text is to be shown normally or in indented outline form, with or without section numbers, with or without format codes, or in ragged-right or right-justified format. **Display keyword** or **password viewspecs** allow the user to see pieces of text only upon the presentation of the proper key; this facility can be used to protect sensitive materials and to reduce the clutter of online presentation by presenting only (potentially) relevant information. Similarly, a syntax-directed editor can allow the user to turn off the display of declarations or comments in a program.

Since the view is a mapping of the internal data structure onto the user's display, external factors such as noise on the communications line or a received system message can corrupt the displayed view, while the internal data structure remains intact. A system often supplies a `refresh display` function to restore the appropriate view from the internal data structure.

3.2.2 Formatting[10]

The simplest method of formatting the filtered viewing buffer contents is the "null" formatting: the text is shown exactly as it is stored in the internal data structure. Typically this is not sufficient for the user. In simple editors, text is stored essentially as consecutive characters on disk, with special characters indicating tabs and physical end-of-lines. A simple formatting routine is needed to map stored lines to a screen image, inserting logical carriage returns and/or linefeeds where appropriate. In many editing systems, especially stream editors, the formatting routines must make on-the-fly decisions about where to break lines that do not fit on one screen line [KNUT79, ACHU81]. For program editors, an intelligent formatting routine might include automatic indentation of program constructs.

In early text processors, **formatting codes** (such as `.pp` for paragraph, `.in 5` for indent 5 spaces, `.ce` for center) were typed in as literal text and subsequently compiled in a batch formatting pass to produce formatted pages; no online feedback was available. Later **softcopy** or **proof copy** facilities were made available to display (but not to allow editing of) monospace (line-printer quality) output on the alphanumeric terminal. With high-resolution point plotting and raster displays, even proportionally spaced typeset text could be previewed outside the editor as the result of a separate batch formatting pass. The next major step in the formatting field was the creation of the interactive editor/formatter. Today most editors, especially commercial word processors, instantaneously display the results at least of commands with local effect (such as `indent`, `tab`, `embolden`, and `center`) as they are entered or changed. In the newest

[10] We do not attempt to cover the entire field of formatting here, but simply point out salient features of formatting directly tied to the editing task. We refer readers to Furuta et al. [FURU82] for a thorough survey.

generations of word processing systems the idea of on-the-fly formatting has been taken to the next logical step: they allow essentially all formatting, including hyphenation and pagination to be done on-the-fly. Interactive editor/formatters make possible an especially useful view in which all operations on the document take place immediately on a displayed facsimile of the printed page, thus giving instant user feedback. Other optional views may include the facsimile document along with ancillary windows in which the typesetting codes controlling the formatting are displayed (see the description of the Xerox Star in Part II, Section 4), or with inline "tags" describing the various document elements that appear on the screen (see the description of Etude in Part II, Section 4).

With hardware such as the personal workstation becoming prevalent, users will be less patient with traditional, tiresome, monospace characters on 80 x 24 CRTs, and will demand that display output make use of the many traditional, hardcopy information-coding techniques that improve reading efficiency. These include proportionally spaced text, font changes, highlighting, complex page layout, and even imbedded equations and graphics. Thus online formatting for display (softcopy) and offline formatting for paper (hardcopy) will become more similar, especially when bitmap raster displays approaching the 200-points-per-inch resolution of inexpensive (less than $10,000) electrostatic or laser printers become commonplace. (Even before this becomes a reality, we see that computerized typesetting, once reserved for special documents or for final copies only of a document whose drafts were produced on line printers or typewriter terminals, is now within the reach of many commercial or academic installations for routine work.) Editors must therefore make it possible for the user to specify sophisticated formatting effects shown online during the editing phase. Good examples of systems providing interactive typesetting can be found in several works in the literature [LAMP78, HAMM81, SEYJ81, SMIT82, and XERO82].

3.2.2.1 Formatting Commands

Approaches to the specification of formatting are numerous. In some systems, primitives provided as part of the interaction language allow the user to specify formatting operations on designated elements. For instance, the use of the `center` command in the editor might actually insert the appropriate spacing for centering and alter the internal representation of the document. Alternatively, this type of specification may be used to generate a low-level representation such as a formatting (typesetting) code that is stored in the file. In some systems these are invisible to the user and a mechanism must be provided to change these low-level representations using high-level editing commands. For example, many word processors provide a **tab rack**, an on-screen simulation of a typewriter's margin and tab controls, that contains the current margin settings, font styles, tab stops, and so on. The user can change these settings by simply moving the cursor into the tab rack and editing the attributes. In Xerox's Star, alternatively, every element (from a single character up to the entire document as a whole) has associated with it an optionally displayable **property sheet**, which simulates a pre-printed form or checklist. The user can examine the property sheet at any time and fill in or change the stored formatting options for any element (see Fig. 6).

In other editors, the formatting specification is entered as a formatting code in the same manner as "normal" (literal) text. In some systems, these codes must be entered on separate lines to distinguish them from literal text; in other systems a special character is used as a delimiter so that the codes can be imbedded in the normal text stream; two delimiters in a row then designate a literal delimiter character.

Regardless of how they are indicated, the specifications are one of two types: **procedural** or

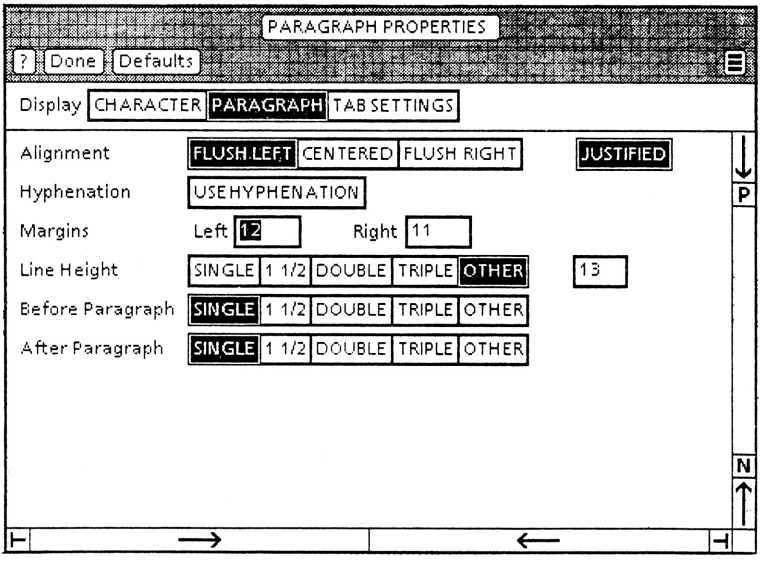


Figure 6. *Xerox STAR Property Sheet.*
The Xerox Star property sheet allows the user to edit attributes of various elements. The paragraph property sheet lets the user change alignment, spacing, and hyphenation by pointing at graphical buttons on the screen. Selected attributes are shown in reverse video.

declarative (markup). In a procedural specification, the author indicates the exact operations to be done to effect the formatting choices (e.g., skip 2 lines and indent 3 spaces). Conversely, in a declarative system, **tags** are used to identify elements of the document, such as items in a numbered list, paragraphs, chapter headings, and running heads. A separate facility stores the actual formatting attributes for the particular elements, and the formatter recognizes the tags and uses the tag facility (usually a set of formatting macros or a database of formatting information) to determine *how* to format any part of the document. These tags allow the same document to be formatted in different styles or for different output devices by simple parameterization: for instance, the user could format a *Computing Surveys* document by specifying the tag `<Style=Surveys>` at the beginning of the document and for *Communications of the ACM* by specifying the tag `<Style=CACM>`. Similarly, the user could format a document for the lineprinter by specifying the device tag `<Device=lpr>` and on a high-quality typesetter by specifying the tag `Device=typesetter>`. **Style sheets** containing a skeleton of tags for a particular document, for example, an online corporate memo that can be filled in by users, allows standardization of documents for a community, much like preprinted stationery or pads. Declarative languages provide less complicated user specifications, and often less complicated

editor interfaces for online formatting; the user need not supply detailed formatting commands, and need use only high-level structural indicators as tags. Reid's SCRIBE [REID80a, REID80b, REID80c] is a popular declarative language and document compiler. IBM's GML [GOLC81] is a formatter-independent declarative language. Hammer et al. [HAMM81], Walker [WALK81b], and Chamberlin et al. [CHAM81] discuss editors that make use of the declarative technique (see Part II, Section 4).

3.2.3 Window Manipulation

In editors with multiple windows, the user must be able to specify and alter the placement of the windows on the display. The system can allocate windows: a single window would occupy the entire screen, two windows, half the screen each, and so on. The system can allow the user to specify the windows, either by typing coordinates or, more likely, by defining the bounds with a pointing device.

3.3 Editing

Our discussion of the editing component focuses on two major concerns: specifying the operands of an editing command and the commands themselves.

3.3.1 Specification of Scope

The range of a document used as an operand for an editing operation is called its **scope**. The user typically thinks of the scope not in terms of "raw" text, but rather in terms of logical elements that can be both edited and traversed. These elements are also known as units, nouns, objects, and structures, and are often a function of the internal representation of the target data. In line editors, for example, manuscript text may be stored as sequential lines and edited and traversed on a line basis. In stream editors, the text may be stored as a one-dimensional, indefinitely long stream of characters, broken into discrete lines by formatting routines for viewing purposes. Programs may be stored in textual form or in some intermediate form such as a parse tree or abstract syntax tree.

In addition to logical elements that match the internal representation, editors provide a variety of elements corresponding to user-level abstractions, often grouped in order of increasing scope: characters, words, lines, sentences, paragraphs and sections are typical for text. In addition to these standard system-provided elements, users can also define arbitrary **regions** or **blocks** that exist only for the duration of a given operation. A third class of elements may be a (partially overlapping) set of abstract document components such as sections, headings, titles, running headers, footers, and numbered lists. These are typically defined just in terms of formatting conventions and are available for explicit manipulation only in the most recent editors.

The goal of scope specification for all these elements is to approximate within the limitations of the computer interface the ways users would manually gesture ("delete *this*, move *that* over *there*"). As we have seen, in line editors the user establishes the current line (by specifying its line number or by traveling to it via scrolling or pattern searching) and then specifies the scope (by typing either a context pattern within that line or an integer to indicate a group of one or more discrete lines whose first line is the current one). In the UNIX *ed* editor the command

```
s/Julius/Brute/
```

substitutes the word "`Brute`" for the scope "`Julius`" when the user is at the line

```
Et tu Julius!
```

In stream editors, of course, the pattern matching is not limited to a current line but has potentially the entire file as its domain. In display editors, the user specifies the scope by driving a cursor with a locator device to define the appropriate element(s), in a manner more directly analogous to the manual technique of pointing.

There are several techniques for specifying a scope, some applicable to a typing-oriented interface, some applicable to a pointing-oriented interface, and some applicable to both.

In the first technique, the user simply selects an element – a character, word, line, or paragraph, for example. The user may **extend** this selection by selecting another element of this same type; all elements between the starting point and the other selected point (inclusive) would be selected. This is typically done using the ellipsis or regular expression facilities in a textual interface, and direct cursor positioning in a pointing interface.

In the second technique, the user adds modifiers to the commands. For example, having established a location in the text with context or cursor specification, the user can complete or modify the scope of a `delete` operation with `word` to form `delete word` or with a numeric parameter to form `delete 3 words`.

In the third technique, the user again establishes a starting selection at the smallest element (for example, a character) and then **adjusts** the selection. For instance, by using the mouse in Xerox's Star, a character can be selected as an initial scope and this scope adjusted to include successively the word, the line, the sentence and the paragraph containing that character (see Figure 7).

In a hierarchical structure editor, selection and adjustment can be easily used to traverse the hierarchy, as in `select node` and `extend to left child`.

3.3.2 Editing Operations

3.3.2.1 Creating

Text is inserted into a computer-based document with a text input device. To enrich this hardware, special software is frequently supplied. Editors often provide automatic **wordwrap**, which eliminates the need for typing a carriage returns at the end of each line. When the editor senses that the word currently being typed has exceeded the right margin, it breaks the line at the first blank space before the overflowing word and automatically pushes it, followed by the cursor, to the next display line.

Display editors generally offer one of two kinds of input styles: **typeover mode** or **insert mode**. In typeover mode, each typed character replaces the character at which the cursor is pointing. For adding characters, such editors supply `insert character` or `insert line` functions that open a blank character or a blank line at the cursor position. In insert mode, each time a user types a character, the character at the cursor position and all those to its right are shifted right one character and the typed character is inserted at the cursor position. Thus insertion can always be done without any commands, while replacement requires a command such as `delete character` or `delete line` before or after the insertion. Many editors allow the user to toggle between typeover mode and insert mode.

Other creation techniques include the capability of **imbedding** files or parts of files into the document being edited, thus making it possible to recycle previously created material. The user can create **boilerplate** documents, as is often done with proposals, contracts, and specifications, by using bits and pieces from the entire domain of user files on the computer. Form letters may be created by imbedding consecutive addresses from an address file at the top of a generic letter file.

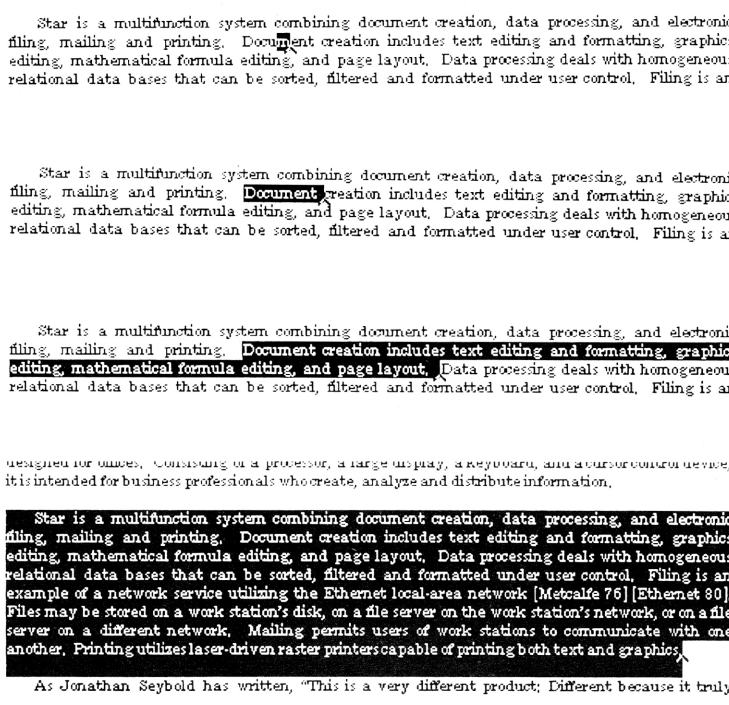


Figure 7. *Selection in Star.*
One click of a mouse button selects a character, the second click, as shown on the left, adjusts the selection to be the word containing that character, and the third click, as shown on the right, adjusts to the sentence containing that word.

3.3.3 Deleting

The **delete** command requires the user only to select the scope of the operation. Since **deletes** are obviously dangerous commands, some systems require confirmation before actually completing the operation. Other systems allow the user to **undo** commands, making the deletion operation reversible. For the **delete** command, as well as the **copy** and **move** commands described below, many systems provide **delete buffers**. This allows the deleted elements to be placed in "limbo" so that they can be used later as the objects of **put** (also called **insert**) operations, which put the elements from the delete buffer back into the text. To move a paragraph, for instance, the user first selects it and specifies the delete operation; the editor then deletes the paragraph and places it in the delete buffer. To put the paragraph back in somewhere

else, the user indicates the desired destination and specifies the **put** operation. Often, systems provide multiple, named delete buffers for more complex manipulations. Berkeley's *vi* editor [JOY80b], for instance, has a **buffer stack**, which keeps track of the last nine pieces of text that have been deleted. EMACS [STAL80, STAL81] has a **kill ring**, a circular list containing the last eight blocks of text that were deleted. The user can "walk around" the previous kills in this ring until the desired text is found, with the walk always leading back to the starting point. Usually delete buffers and kill rings are saved while performing inter-file editing, so that text deleted from one file can be inserted into another easily.

3.3.4 Changing

Many of the changes made to a document are corrections of typographical and other minor errors. The simplest change is the replacement of one letter with another. In typeover mode of a display editor, the correction is made by simply typing the new character over the erroneous one. In insert mode of a display editor, the correction is made by typing the new character and then using a `delete-char-forward` command to delete the old character, which has been pushed to the right by the insertion of the new character. Similarly, changing a word in typeover mode simply involves typing over the erroneous word. However, since the replacement word may be shorter (longer) than the original word, a `delete (insert)` character function may also be needed. In insert mode, changing a word would be done entirely with the implicit insertion combined with `delete-word` functions.

In editors without cursor keys, the user needs a method of specifying what character(s) to replace. These editors have a **change** or **substitute** command, as shown previously, that takes as arguments both the scope of the change and the replacement string.

Because of its speed, the computer naturally offers the facility of global operations – operations that take place uniformly throughout a document. The **global change** command allows the user to specify a pattern to be found throughout a document and a replacement string to replace that pattern wherever it appears. In some systems, this command may do its work but not indicate what changes have been made. In others, a count of the number of changes is given. In others the pattern string is highlighted each time it is found, and the user is prompted to indicate whether or not the change is desired. Column-dependent changes are useful for editing programs or tables.

The **transpose** command is a special-purpose change command. In EMACS for example, the CTRL-T command will exchange the character at which the cursor points with the one directly to its left, and similarly, the META-T command will do so for words.

3.3.5 Moving

The ease with which blocks of text can be rearranged is one of the great advantages of interactive editors. To **move** a block of text the user specifies a source (the scope of text to be copied) and destination (the place where the text is to be copied). In an editor in which the scope is specified by pointing, the user defines the source by selecting the beginning and end of the text to be moved, and then defines the destination by pointing to the location at which it should be placed.

In line editors and context editors, where such a physical sequence of steps is impossible, the user must resort to textual specification. For example, the misordered poem by Frost

```
Two roads diverged in a yellow wood,
And looked down one as far as I could
To where it bent in the undergrowth;
```

```
And sorry I could not travel both
And be one traveler, long I stood
```

could be ordered properly with a typical line editor by traveling to the line "`And looked down one as far as I could`" and specifying

```
move 2 down 2
```

This would temporarily delete the following two lines, including the one currently being pointed to, move the current pointer to the line following the last line deleted, move it down 2 more lines, and insert the temporarily deleted lines below where the current pointer now points.

Note that with most line editors, the user is severely limited to moving integral numbers of lines, rather than arbitrary regions. In a context-driven stream editor this specification is a bit easier:

```
move/And looked...undergrowth;/stood/
```

Rather than specifying relative line numbers, the user now specifies context patterns.

Note that systems with delete buffer facilities may have no need for a special-purpose move command, since, as discussed in the previous section, one can delete the desired text and re-insert it in the appropriate place with a `put` command.

3.3.6 Copying

A `copy` command is simply a `move` command in which the source text is not removed after the destination text is in place.

An alternate strategy for the `copy` command is to have a `pick` command that stores a selection of text in a **pick buffer**. This is analogous to using the delete buffer described above, with the important exception that the picked text is not deleted. Now a copy of the picked text is in limbo in the pick buffer and can be reinserted in the text with the `put` command. If the `put` command does not erase the contents of the pick buffer, this facility can be used to make multiple copies of some selected text quickly.

3.4 Miscellaneous Capabilities

Commands grouped under this heading are not directly involved with manipulating text, but rather with assuring the integrity of user's work and with making the user interface more powerful and helpful.

3.4.1 Reliability

3.4.1.1 Backup Capability

To minimize the possibility of the accidental erasure or destruction of a document, editors often have **backup** capabilities. One strategy is to give the user **workfiles**, copies of the actual files, to work with, saving them as the actual files only when the user exits or when an **autosave** feature copies the workfiles as the actual files after a (user-specifiable) number of keystrokes or command executions. A similar strategy is to make automatically a backup copy of the files as they exist when the editor is invoked; the user is given the actual files to edit, but can always make the backup copy become the actual file in the case of a system crash or editing mishap while editing the actual file. An `abort` command allows the user to scrap an editing session and return to the backup copy that contains the state of the file when the editing session began.

3.4.1.2 Undo Facility

The **undo** facility is a critically important, time-saving, and unfortunately not yet universal feature. The most basic version allows the user to undo the last command entered. More

useful systems have an *n*-level undo stack that allows the user to undo commands *n* levels back (sometimes to the beginning of the session or even back to previous sessions). The undo feature frees the user from the burden of making sure that each command does exactly what is wanted by guaranteeing that any result can be undone. The general undo facilitates risk-free experimentation, especially important for online composition and organization. Some systems [ARCH81] provide an undo-redo facility, which allows the user to undo operations and then redo them at the push of a button. Another way of presenting **undo**, rather than as a complete backtracking of a session, is simply as a command that performs the inverse of the last specified command. For instance, after one performs a **delete**, the **undo** command would essentially perform a **put**. Issuing **undo** again would then perform a **delete**. Smalltalk-80's **cancel/accept** pair allows the user to **accept** the editing state at a point in time and experiment freely, knowing that a single **cancel** command will return the document to the accepted state.

3.4.1.3 Cancel Facilities

The **cancel** command allows the user simply to cancel the command that is currently in progress. This is of great importance if time-consuming operations such as pattern searching have been specified erroneously. When commands are not queued and typeahead is not in effect, specifying a new command while a previous command is still executing often implicitly cancels the executing command. Alternatively, especially when an operation is simply a traveling command rather than a command that makes extensive changes to the internal data structure, specifying a command while one is executing could cause feedback from the first command to be cancelled. For example, if the user presses the + PAGE key in a screen editor, and, while the new page is being displayed, the user presses + PAGE again, an intelligent system would not first complete the display of the first page and then replace that completely with the second page, but would cancel the display of the first page and proceed to display the second page.

3.4.1.4 Keystroke History

The **keystroke history** is a powerful reliability mechanism that keeps a copy of every keystroke (both command and text) that the user has specified since the current editing session started. If the system crashes, the user is provided with mechanisms to run the keystroke history file against the old copy of the edit files as if the commands were being typed in. This history file can also provide the basis for the undo command.

3.4.1.5 Context Saving

A useful feature provided by some systems is the ability to save editor attributes and parameters from session to session. In *bb*, for example, a control file (see Fig. 8) saves information about the initial cursor position when the editor is invoked, the file to use, and enough other information that the user can simply invoke the editor, without any parameters, and continue a session exactly where it ended five minutes or five days ago. In more advanced systems, a **checkpoint/restart**, **snapshot** or **workspace** facility allows the recording and subsequent restarting of the entire editing environment, including complete internal data structures. This is common in programming languages like LISP, APL, and Smalltalk-80.

3.4.2 Ergonomics

3.4.2.1 Repetition

A **repeat** command, which re-executes the command last executed, is a simple and convenient facility. Another alternative is to have the user apply a new selection and then reissue the command. Another is to "bring back" the last command and allow the user to edit it before re-executing it.

```
savetemp=1
bufinfile=.bbbuf
bufoutfile=.bbbuf
filename=macrofig.me
curcrypt=0
curstream=1
curtabin=0
curtabout=0
curindent=1
curline=19
curchar=43
curscreen=0
curmargin=0
curwindow=0
curreadonly=0
curlanguage=roff
fileid=10
Pfilename=dopstitle.tex
Pcrypt=0
Pstream=1
Ptabin=0
Ptabout=0
Pindent=1
Pline=17
Pchar=6
Pscreen=7
Pmargin=0
Pwindow=0
Preadonly=0
Planguage=
Pfileid=10
Qfilename=/usr/info/b.help
Qcrypt=0
Qstream=0
Qtabin=0
Qtabout=1
Qindent=1
Qline=0
Qchar=0
Qscreen=0
Qmargin=0
Qwindow=0
Qreadonly=0
Qlanguage=
Qfileid=2
Asearch=shar
Arsearch=^[WM]
Afsearch=
Aexit=
Abfsave=
```

Figure 8. *A bb Control File.*

The control file saves information from one editing session to another. Here, for example, the `bufinfile` and `bufoutfile` variables indicate where the pick and delete buffers are to be saved from session to session. `Curline` and `curchar` indicate where in the document the cursor was pointing when the session was ended. `Curscreen` indicates which line is at the top of the screen. `Filename` indicates what file was being edited at the time the session was ended. The variables beginning with `P` and `Q` keep track of the same information for multiple files.

3.4.2.2 Online Documentation/Help Facility

An online **help facility** is extremely important to new and occasional users, as well as dedicated users who do not use all parts of the system regularly. The help facility can provide an expanded explanation of an error message, a short summary of a command syntax, or perhaps complete access to an online version of the manual. Some systems create a separate help window, allowing the user to have access simultaneously to both the help information and to the document being edited, rather than forcing the user to leave the document and lose the information on which the user sought help in the first place.

3.4.2.3 User Feedback

Feedback is a vital part of the editing process; it is necessary for specifying operations, for specifying scopes, and for showing the results of an operation in the updated view. The last kind of feedback is provided out of necessity, but the first two categories are not always given due consideration by editor designers.

In editors with typing-oriented interfaces, echoing the typed command provides immediate feedback on both operation and scope. In function-key interfaces, a button push provides no inherent feedback. If no supplementary feedback were supplied, the user would have to rely on examining the results of the operation to see if the specification was correct; by this time, it would often be too late to reverse the results. Thus feedback techniques, such as highlighting a selection in progress (with such hardware-supported techniques as brightening, underlining, or

reverse video) in display editors, or highlighting the menu items as they are browsed through in menu-oriented interfaces, are vitally important. In non-display-oriented systems, a summary of the command that is to be executed or has been executed is useful. This feedback might consist of displaying a condensed English-language message such as "`move 'red fox...dog' after 'The quick'`." If an editor has undo facilities, this type of condensed feedback of results is useful but not necessary, since the user can afford to make mistakes or experiment.

Other audio and visual cues aid the user at a little cost in efficiency or in implementation. Beeping to signal errors usually ensures that a user does not miss the occurrence of an error. Programmable cursors allow the cursor to take on different symbolic forms depending upon what the user is doing. Time-consuming operations require intermittent feedback so that the user can be satisfied that the system is still working. Newshole [TILB76] uses a Buddah to remind the user to be patient while the Xerox Star uses an hourglass icon to indicate that it is busy. A status line at the top or at the bottom of the screen, indicating the current position in the document, the name of the file, any modes that might be set, and other such information, is an easily implemented method providing positive feedback.

3.4.3 Customization

3.4.3.1 Profiling

A **profiling** facility allows the user to "tune" the editor environment at invocation time. This allows important or preferred environment settings to be handled automatically and removes the need for all users to accept a common default. Some editors allow the setting of simple state variables such as number of backup versions and number of modifications before saving the file on disk. Others allow each user to reconfigure major parts of the interface, such as redefining function key bindings.

3.4.3.2 User-Defined Commands (Macros)

Editing systems often allow the user to define **macros** or **editing scripts** (super-instructions) based upon the system operation repertoire using an editor **macro language**. The user can thus package under one name sequences of commands that are often executed as groups (see Figure 9). In some systems, these commands prompt for or simply accept parameters (operands) and even provide conditional execution, for maximum power and flexibility. Function-key editors often have **keystroke macros**; the system "captures" a set of keystrokes typed in by the user and can then repeatedly execute those keystrokes as if they were one command. Some keystroke macro systems even allow the user a form of parameterization, temporarily stopping the execution of the keystroke macro, allowing the user to type in the "parameter," and then finishing the execution.

3.4.3.3 Extensibility

Some editors allow the user to extend the command set, in the same language in which the editor is written. Thus the user is not limited to designing macros made up of editor primitives, but can design operations using the same lowest-level primitives as the nucleus of the editor uses itself. The fact that the extension is being done in an actual programming language, rather than a special-purpose editor macro language, implies greater efficiency and ease of expression of new functions. In EMACS, for instance, the editor can be used to modify or to create a function and this function can be "linked" into the editor without ever leaving the editing environment. Of course, such a feature is not targeted to the general public but to more advanced users or programmers who are willing to learn the internals of the editor to modify or to add code.

Suppose a user has mailbox file which contains all the headers and text of messages received for a period of time and would like to create a shorter version of this file for quick reference containing only the headers. A macro to do this might look like this:

```
DEFINE MACRO mail_to_headers
deluntil /From /
skip 2
insert ---------------------------------------------
next
DEFINE MACRO END
```

In this hypothetical editor, the user defines a macro called *mail_to_beaders* by bracketing typical editor statements between a *DEFINE MACRO* and a *DEFINE MACRO END* bracket. The statements in this macro indicate to delete everything until the first line with the characters "From ", to skip 2 more lines (the "To" line and the "Subject" line in the mail message), to insert a line of underscores, and to go to the line after these new underscores.

Given, with the current line pointer pointing to a line preceeding the line "From wcs...":

```
From wcs Thu Mar  5 02:49:41 1981
To: nkm
Subject: (un)natural language processing
Cc: wcs

hl nrmn.

ths is a tst of th nw unix* cmnd avd. It mr  or ls
bindy strps vwls frm stndd inpt & pics th rslt
on stndd otpt.
  .
  .
  .

From avd Fri Mar  6 20:17:16 1981
To: nkm
Subject: Another lost cause
Cc: skf wp

Do you know where the excess fress resource manuals are, or the master
for that matter? Do you know how to copy it off the C-disk so someone
could take a look at it? Where the source is kept these days?

From skf Fri Mar  6 18:47:14 1981
To: fac grad graphics ugrad
Subject: The ACM Lecture Series: Judson Rosebush

On Thu Mar 12 @ 4pm in Pembroke Hall 210, the ACM will be sponsoring a
lecture by Judson Rosebush, president of Digital Effects, a NY-based
company doing commercial computer animation.
will be shown on film/videotape. The talk should be
```

successive invocations of the macro *mail_to_beaders* would result in the file being changed to:

```
From wcs Thu Mar  5 02:49:41 1981
To: nkm
Subject: (un)natural language processing
---------------------------------------------
From avd Fri Mar  6 20:17:16 1981
To: nkm
Subject: Another lost cause
---------------------------------------------
From skf Fri Mar  6 18:47:14 1981
To: fac grad graphics ugrad
Subject: The ACM Lecture Series: Judson Rosebush
---------------------------------------------
```

Figure 9. *Example of Macro Facility.*

3.4.4 Target-Specific Operations

Target-specific operations are those not common to all editors but specific to the target application area for which the editor is designed. A LISP program editor, for example, might have routines to locate matching parentheses. The target-specific operations are the most marked distinguishing characteristics between editors. These operations go further than manipulating elements as simply neutral parts of some larger whole; rather, they operate on these objects with some "knowledge" of what they are, for example, a `capitalize` command would need to know that a capital **a** was represented as an **A**. In syntax-directed program editors, for example, compilation of a syntactic entity is a target-specific operation. A more thorough description of these will be found in the next section in the discussion of actual editors.

PART II: SURVEY

4. IMPLEMENTATIONS

This survey discusses a wide variety of editors used in academic and commercial circles. Our purpose is not to provide a detailed point-by-point comparison; our coverage from editor to editor is not necessarily consistent in either subject matter or depth. Rather, using the terminology of our tutorial, "Interactive Editing Systems: Part I," we attempt to illustrate the capabilities outlined in Part I, Section 3 of that tutorial by briefly describing the distinctive features of each editor or class of editors. While a taxonomy of the interactive editor – one in which we could compare the genealogy, purposes, and features of various systems – would be useful, it is difficult to construct. Terminology for categorizing editors is far from standard, a fact that often leads to identical labels for less than identical software and hardware. The history of editing contains many parallel developments and much cross-fertilization of ideas; a strict ordering or categorization is thus impossible. Informally, then, we will be looking at editors from the viewpoint of the target applications for which they were designed, the elements and their operations, the nature of the interface, and the system configuration. These categories do not form strictly independent axes; the choice of one frequently influences the choice of another.

"Target applications" are the high-level entities that the editor manipulates, for example, manuscripts, programs, or pictures. "Elements" are the units of target data that may be manipulated by the user. For example, a user may manipulate a program in the units of single lines of text, of individual programming language constructs, or of individual nodes in a parse tree. User "operations" fall into several subcategories. Editing operations allow the user to manipulate the target elements. Traveling operations allow the user to browse through a document. Viewing operations allow the user to control what subset of target data is presented to the user and how it is formatted: for example, text may be viewed as single lines, as full-screen pages, as a prettyprinted program, or as a facsimile of a typeset document. "Interface" defines the interaction language, input devices, and output devices with which the user performs these operations. "Configuration" describes the architecture of the systems on which the editor can run.

For compatability with popular terminology, we review some of the most common terms that will be used in the discussions in this section. A **text editor**[11] is one of the basic components of a **text processing system**, which is concerned not only with creation and maintenance but also with formatting and interactive presentation of text. In addition to a text editor, a text processing system includes a **text formatter**, concerned with the layout and typography of the text, and various **text utilities** such as spelling correctors that aid in analyzing and preparing the text. **Word processing** is a commercial synonym for text processing. An **office automation system** typically combines a word processing system with utilities such as database management, information retrieval, electronic mail, and calendar management. **Program editors** operate on programs, whether represented in textual form or in another canonical form, such as a parse tree or abstract syntax tree. **Picture** or **graphics editors** facilitate the creation and revision of computer-based graphics. A new development in the text processing field is the **document preparation system**, which integrates text editing, picture editing, and formatting. A **voice editor** is a specialized interactive editor in which the target is digitally-encoded voice.

[11] In this paper, Bold type is used to introduce concepts and terms. Typewriter type is used to set off editor commands. Italic type is used for emphasis.

A **forms editor** is an interactive editor that allows users to create and to fill in business forms conveniently. An **interactive editor/formatter**, often called a "what-you-see-is-what-you get" editor/formatter, allows the user to edit a facsimile of the printed page such that the changed text is reformatted instantaneously. On standard alphanumeric terminals, the facsimile represents a monospaced, typewriter page. On high-resolution raster graphics displays, the facsimile represents a proportionally-spaced, typeset page, with a variety of typefaces, sizes, and weights, and such non-textual material as equations, line drawings and even photographs. The goal of the **universal** or **virtual** editor, a current topic of research, is to generalize and integrate previously target-dependent software, providing a uniform way to manipulate seemingly dissimilar targets such as manuscript text, program text, pictures, and digitized voice. A **structure editor** is a special type of virtual editor that gains its generality by imposing the same structure on different targets. For example, a structure editor based on hierarchy may allow the user to impose this tree structure on diverse targets and edit them with the same tree-editing primitives (e.g., `delete subtree`, `move current subtree up 1 level`, `display all siblings of node`). A **syntax-directed editor** is based on the same principles as a structure editor but imposes the syntactic structure of a particular language, rather than a general-purpose structure, on the target.

4.1 Line-Oriented Editors

Line-oriented editors are covered here simply to round out our treatment of common editors. We do not, however, advocate the continued production or use of these editors. The conceptual model presented with line editors is that of editing virtual card images; the line editor constantly visits the limitations of this outdated representation of data on the user. Notable drawbacks are pattern searches and edits that do not cross line boundaries, and overflow and subsequent truncation of fixed-length lines. The continued dependence on the card analogy illustrates an important design flaw in many editors: they adhere to outmoded conventions even though those conventions unnecessarily limit the technology of the day. Unlike the TECO model below, where lines are simply an optional filtering presented to the user as a service, line editors force this limited view on the user.

IBM's CMS Editor

IBM's CMS editor (c. 1967) is a classic example of a fixed-length line-oriented editor with a textual interface, designed for a timesharing system whose terminals lack cursor motion keys and function keys. It presents the user with a one-line editing buffer (the amount of the document that can be edited at a given time), though this is extended for some operations. Similarly, it presents a corresponding one-line viewing buffer (the amount of the document that is used to construct the display). The display is a simple mapping of the one-line viewing buffer to a one-line window; it is typically updated after the execution of each command. (A more thorough explanation of the editing buffer, viewing buffer, and window model is presented in Part I, Section 1). Traveling is done with line granularity using absolute and relative **gotos** to varying internal line numbers and context pattern-matching. The input language is textual with two major modes: input mode and edit mode. Typically the user spends most time in edit mode, with input mode reserved for bulk input of text. The prefix syntax is generally consistent across commands:

command/scope/optional destination/optional parameters

The commands are full English words; the user does not have to remember abbreviations, though the system will accept the smallest possible unambiguous abbreviation. Most commands operate upon the line units, and across lines, as well, if so specified by the scope.

We now show some simple editing using the CMS editor. Assume we are in edit mode and that the following section of a program, which computes the sum of two matrices, is to be modified to compute the difference of the two matrices:

```
ADD: PROCEDURE;
FOR ROW = 1 TO N DO;
  FOR COLUMN = 1 TO M DO;
    C(ROW,COLUMN) = A(ROW,COLUMN) + B(ROW,COLUMN);
  END;
END;
```

The following sequence of interactions with the editor would provide the necessary changes (the user's requests are preceded by the system prompt character ">".

```
>find/add:
ADD: PROCEDURE;
>change/add/subtract
SUBTRACT: PROCEDURE;
>next 3
    C(ROW,COLUMN) = A(ROW,COLUMN) + B(ROW,COLUMN);
>change/+/-
    C(ROW,COLUMN) = A(ROW,COLUMN) - B(ROW,COLUMN);
>top
>change/add/subtract/ * *
```

The routine ADD is first located by using the column-dependent **find** command that searches for the string "ADD:" beginning in the first position of a line. (The locate commands, after searching for a pattern that does not exist, travel to and leave the buffer either at the beginning or end of the file, frustrating the user who has erroneously specified a search pattern and must manually grope back to the former location.) The current line pointer now points to the line "ADD: PROCEDURE"; this line is echoed on the screen. The next user command, **change/add/subtract**, affects only the contents of the buffer: the first occurrence of "ADD" is replaced by "SUBTRACT." For appropriate types of files, the editor does automatic lower-case to upper-case translation. If the maximum line length of 132 characters is exceeded, the editor will truncate the line. Line numbers in the CMS editor are varying. Travel is specified relatively, with **next** (**next 3** moves the current pointer and hence the editing and viewing buffers 3 lines down from the current location), or absolutely, with **goto** (**goto 276** moves the current pointer to the 276th line of the document). The **change/+/-** command changes the "+" to a "-". The **top** command moves the current line pointer to the first line of the file. The "* *" operand of the final **change** specifies the replacement of *all* occurrences of "ADD" in *all* lines – this is a global change that will affect the entire file.

The CMS editor provides the ability to set up logical tab stops – tabs implemented in the editor software rather than in terminal hardware – so that tabs may be specified by typing a user-chosen logical tab character in the input stream. Certain installation-specific enhancements of the basic CMS editor allow the user to undo the most recent command, shorten the scope specification by using ellipses (. . .), and do automatic indentation tailored to language-dependent needs [BROW81].

One of the most confusing attributes of the CMS editor are its two modes. Edit mode gives the user access to all the functional capabilities of the editor, including the capability to switch

to input mode. Input mode, though, only gives the user two options: typing in text, which is simply inserted into the file at the current line pointer, or pressing dual carriage returns, which returns the user to edit mode. (A blank line is entered by typing at least one space). Even if the text that the user types in input mode is a command, it is not executed. To get into input mode, the user types the command **input** while in edit mode (if the file is new, the user is automatically placed in input mode on invocation of the editor). Often, a user might type a sequence like

```
locate/bull
next 3
type
```

only to discover, after some pondering, that the system is in input mode and that these commands were not executed but were actually inserted into the file as text. The "fix" is to get into edit mode, move the current pointer **up** *n* lines to the first erroneous line, **delete** *n* lines, move **up 1** to reposition the pointer to the location from which the erroneous commands were first issued, and finally reissue the commands as commands.

SOS

SOS [DIGI78], like the CMS editor, is a line editor designed for editing on "glass teletypes" – display terminals underutilized as hardcopy terminal emulators – on a timesharing system, specifically a wide range of Digital Equipment Corporation computers. The input language is textual and is very similar to the CMS editor. The commands, as shown in Figure 10, are typed in prefix notation (verb/noun). The major unit of manipulation is the line.

Unlike the CMS editor, SOS attaches *fixed*, visible line numbers to each line in a file being edited. Typically a file is stored with these numbers, but special commands allow it to be stored without numbers and enable the numbers to be regenerated at the beginning of the next editing session. The editing buffer defaults to one line, though for most SOS commands a user can specify a line number or range of line numbers to expand this editing buffer. The default viewing buffer is a line; the window is simply a mapping of this line to the output device.

For selection and organization purposes, SOS goes one step further and allows the user to create logical pages within a file, using the **page mark** command. This essentially divides the file into subfiles that are independently sequence-numbered. SOS maintains a **current position** pointer made up of the current page number and the current line number.

SOS is a highly modal editor, with the following seven different modes of operation (see Figure 11):

- *Input mode*, in which SOS accepts the text being typed and inserts it into the file;
- *Read-only mode*, in which a user can travel through a file but not modify it;
- *Edit mode*, in which the user spends much of the editing session, performing editing, traveling, and viewing operations;
- *Copy-file mode*, in which the user can copy part or all of ai file into another one;
- *Alter mode*, in which a user can perform character-by-character intraline editing without pressing **carriage return** to execute the command; it is a textual approximation to display editing without cursor keys;
- *Alter/insert mode*, in which the user can insert characters such asiii control characters that have special meaning to the editor; and
- *Decide mode*, in which the user can make case-by-case decisions for **substitute** commands. In fact, decide mode has two submodes, *decide alter* and *decide alter/insert*. These two modes

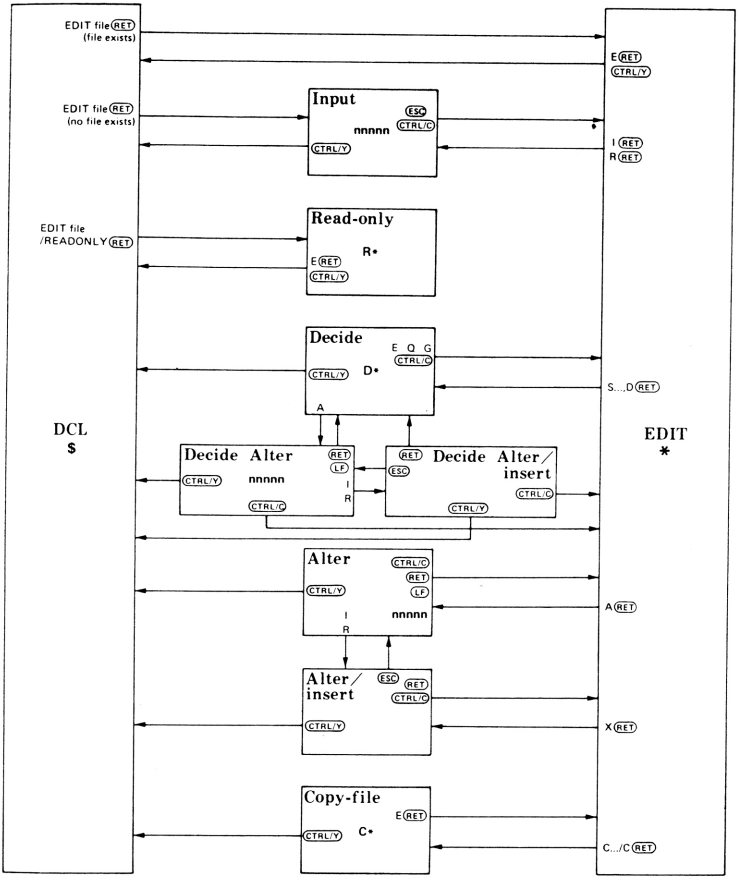
Command	Page	Form	Arguments
Alter[1]	2-27	A	[range]
Copy[2]	2-40	C	position[=file-spec],range[,increment1[,increment2]] position=file-spec/C
Delete	2-43	D	[range]
End	2-44	E	[B][Q][S][T][:file-spec]
Overwrite input file		EB	
No output file		EQ	
Strip line numbers		ES	
No numbers, no pages		ET	
Find[3]	2-46	F	[[string](ESC)[range]][,A][,N][,E][,n][,-]
Help	2-51	H	[:n]
Input[4]	2-52	I	[position][,increment] [position][;increment] [position][:!n]
Join	2-55	J	[position]
Kill page mark	2-56	K	/page
List:LP or file	2-57	L	[range[,S][,P[:file-spec]]] [range][,[S][,F:file-spec]]
Mark	2-58	M	[position]
reNumber	2-59	N	[increment][,[range][,start]]
Print: terminal	2-61	P	[range][,S]
Replace[4]	2-63	R	[range][,increment] [range][;increment] [range][;!n]
Substitute[5]	2-65	S	[[oldstring(ESC)newstring](ESC)[range][,D][,N][,E]]
Transfer	2-69	T	position,range[,increment1[,increment2]]
Save World	2-73	W	[B][:file-spec]
eXtend[6]	2-74	X	[range][,N]
Move Position	2-76	.	position
Give Parameter	2-77	=	parameter
Set Parameter	2-78	/	parameter[:n]
Command File	2-79	@	file-spec
Print next line		(RET)	
Print previous line		(ESC)	

[1] Enter Alter mode.
[2] Enter Copy-file mode (if /C).
[3] Enter Alter mode (if ,A).
[4] Enter Input mode.
[5] Enter Decide mode (if ,D).
[6] Enter Alter/insert mode.

Figure 10. *SOS Commands [DIGI78].*

differ from *alter mode* and *alter/insert mode* primarily in that they, upon returning from the submodes, leave the user in *decide* mode rather than *edit mode*.

Alter mode is the most unusual of the modes. It simulates the intraline editing that is easily provided on display editors, and provides access to elements other than lines. The command syntax is postfix (noun/verb) and infix (noun/verb/noun), not prefix. Commands allow the user to skip forward and backward by characters and words, delete characters and words, capitalize



The paths among the various SOS modes and submodes of operation are marked by arrows. Prompts are shown in boldface type.

Figure 11. *SOS Modes [DIGI78].*

nd uncapitalize characters, delete all characters until the occurrence of a particular character, nd so on. Unfortunately, the user must explicitly enter *alter mode* to take advantage of these acilities.

As Figure 11 shows, the transitions from mode to mode are almost mazelike; the user can asily become trapped in a remote area of the system. For instance, in *decide alter/insert mode* 3C brings one to *decide alter mode*, typing carriage return brings one to *decide mode*, typing

Con-struct	Internal represen-tation	Meaning
Find-string constructs:		
?/	CTRL/T	Match any character
?:	|	Match any separator
?<	CTRL_	Match a space or tab
?%x	CTRL/E	Match any character except x
?)x	CTRL/N	Match 0 or more of the character x
?1x	CTRL/V	Match 1 or more of the character x
?9	CTRL/X	Match any alphanumeric character
?!	CTRL/A	Match any letter (A-Z, a-z)
?&	CTRL/F	Match any uppercase letter (A-Z)
?2	CTRL/W	Match any lowercase letter (a-z)
?+	CTRL/P	Match any decimal digit (0-9)
?>	CTRL/J	Match beginning or end of line
?7c	CTRL/G	Match internal representation of c
Substitute-string constructs:		
?"	CTRL/B	Substitute next string matched
?*n?*	CTRL/O	Substitute nth string matched

Figure 12. *SOS Regular Expression Meta-notation [DIGI78].*

CTRL-C brings one to *edit mode*, and typing CTRL-Y brings one to *DCL*, the operating system command interpreter. In *decide alter mode*, these command bindings change. While CTRL-C and CTRL-Y remain the same, now carriage return or linefeed bring the user to *decide mode*, and both I and R bring the user to *decide alter/insert mode*. In decide mode CTRL-C and CTRL-Y still perform the same, but E, Q, and G also bring the user to *edit mode*. This time A, as opposed to ESC, will bring the user to *decide alter mode*. The remaining transitions, as shown in Figure 11, are no less inconsistent and confusing.

Not only are the mode transitions difficult, but the actual command mnemonics for similar commands differ substantially from mode to mode. For example, in *edit mode*, the **f (find)** command allows the user to search for and move the editing buffer to the first line that contains a specified pattern; the **s (substitute)** command allows the user to replace an occurrence of an old pattern with a newly specified pattern. In *alter mode* the **s** now stands for **skip** and allows the user to find the next occurrence of a specified character; **c (change)** allows the user to change the next *n* characters in the line; **f** no longer exists.

SOS has some interesting concepts: powerful scope specification as a suffix to commands; a regular-expression pattern-searching facility as shown in Figure 12; a query-replace user-dialogue set up by *decide mode*; user-selectable toggles to indicate the level of experience of the user and to control the verbosity of prompts, and more. Yet the sheer complexity of the user interface

often makes the system undesirable for even the most dedicated of users. We feel that SOS is a classic example of a powerful nucleus crippled by a poor user interface.

UNIX ed

The UNIX text editor, *ed* [KERN78a, KERN78b], is a variable-length line editor similar to both the CMS editor and SOS. *Ed*'s commands, like those in SOS, are only one or two characters long. It has a single-line viewing buffer but, like SOS, *ed* allows the user to expand the editing buffer for a command by specifying a range of line numbers in the form **starting,ending** as an optional prefix to each command. Thus, to perform the above change from ADD to SUBTRACT on the first 50 lines of a file, we use the *ed* **substitute** command:

```
1,50s/add/subtract/
```

The special **metacharacter** "$" indicates the last line of the file. Thus prepending a "1,$" to a command causes the buffer for that command to be the entire file. To move a number of lines, we simply say

```
1,10m/insert after this/
```

This will move lines 1 through 10 to follow the first line in the document that contains the string "insert after this". Lines in *ed* are variable-length so that truncation problems are solved.

A powerful feature of *ed* is its facility for user-specified regular expressions in patterns defining the scopes of operations (as opposed to other editors which use regular expressions simply for **search** commands). (This feature has been available since NLS, but has become more common in other editors since its implementation in *ed* and SOS.) The user is supplied with the metacharacters

```
* $ ^ .  [ ] \
```

with which to form regular expressions specifying the content of the pattern. The "*" is a repetition character. Thus, a character "n" followed by a "*" tells the editor to match the first character string containing an "n" followed by zero or more occurrences of "n". The "$" metacharacter in this use matches the end of the line while the "^" caret companion matches the beginning of the line. A "." matches any character. The "\" escape character allows one of the metacharacters to be used as an actual character. Finally, the "[ ]" pair allows the user to specify a range of characters to be matched: [a-j] would match the first string (a single character) containing one of the letters lower-case "a" through lower-case " j"; [nkm] would match the first string with either an n or a k or an m. If the user wanted to find the first line beginning with a capital letter followed by a vowel in the text of the Ogden Nash poem

```
I think that I shall never see
A billboard lovely as a tree
Perhaps unless the billboards fall,
I'll never see a tree at all.
```

the user would specify the search (using / as the find command)

```
/^[A-Z][aeiou]/
```

The ^ requires the pattern to be matched to start at the beginning of the line; the [A-Z] requires the first character of the pattern to be matched to be a capital letter; the [aeiou] requires the next character of the pattern to be matched to be a lowercase vowel. Upon executing this command from the top of the file, *ed* would find (and move the current pointer to)

```
Perhaps unless the billboards fall,
```

One interesting feature in *ed* is the ability to reference the scope of an operation indirectly in another operand of that operation. For instance, to parenthesize the entire line above, one would type

```
s/.*/(&)/
```

The ".*" is metanotation that means "match all characters on the current line." The "**&**" is metanotation that is shorthand for "all that were matched."

As in the CMS editor, lines in *ed* have varying internal numbers. Thus traveling is done as in the CMS editor, with both absolute or relative specifications (as well as context pattern specification). A timesaving feature is the use of a simple carriage return in edit mode (with nothing else on that line) as an implicit **next 1** command. The user is given an explicit symbol called the **dot** to reference the current line pointer that can be used in arithmetic expressions to change the scope of an operation. For example,

```
.-10,.+7p
```

tells *ed* to print the 10 lines before the current position, the line at the current position, and the 7 lines after the current position. *ed* also allows the user to mark a specific line with a single lowercase character for later reference. The user simply types the **save position** command (abbreviated **k**) followed by a single character label, as in

```
kx
```

and the current line is now referenced with "**x**." To travel to that line, the user simply types "**'** ", the **goto saved-position** command, followed by the label:

```
'x
```

and immediately is returned to that saved position. Like the CMS editor, *ed* has two main modes, edit mode and append mode, and the associated problems of two modes. In fact, these problems are compounded by the fact that *ed* is, as characterized by Norman [NORM81], "shy":

> Ed's major property is his shyness; he doesn't like to talk. You invoke Ed by saying, reasonably enough, "ed." The result is silence: no response, no prompt, no message, just silence. Novices are never sure what that silence means. Ed would be a bit more likeable if he answered "thank you, here I am," or at least produced a prompt character, but in UNIX, silence is golden. No response means that everything is okay; if something had gone wrong, it would have told you.

In the edit/append mode dichotomy, this silence causes major confusion. To add text to the file, the user issues the "**a**" command to **append**, followed by a carriage return. Unlike the CMS editor, *ed* gives no indication that it is now in append mode; it just waits for the user to input text, like the CMS editor. To return to edit mode, the user types a line with only a "." on it and follows it with a carriage return. As Norman points out, this is not an oversight, but in fact is acknowledged, rather flippantly, in the documentation [KERN78a]:

> Even experienced users forget that terminating "." sometimes. If ed seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.

One of the designers of UNIX system software defends the terseness of UNIX commands by citing their contribution to an important capability of UNIX: the ability to easily use the output of one program as the input to another [LESK81]. But silencing a user-oriented interactive program so that its output may be used by another program seems to us a large price to

pay. In fact, UNIX easily allows the user to select just which output should be passed on to another program as standard input; careful programming can ensure that user prompts and status information can be interspersed with standard output without interfering with it.

While *ed* is a powerful line editor, it is questionable whether the interface, which requires the user to memorize small, non-mnemonic, and often obscure command names and, more critically, to "guess" the status of the system, is proper for a general-purpose audience. In fact, this editor was developed not for a large community, but for a group of a half-dozen computer science researchers familiar with the notions of regular expressions and file organization who were designing the operating system and file system in which the editor would run; they wanted maximum keystroke efficiency and minimum distraction. While the *ed* line editor has illuminated several important concepts in editing, it nevertheless represents a decreasingly popular breed of editors.

4.2 Stream Editors

Stream editors act upon a document as a single, continuous chain of characters, as if the entire document were a single, indefinitely long character string, rather than as fixed-length or variable-length lines. By doing so, they avoid line editor problems such as truncation and inability to perform interline searching or editing. TECO, described below, is the most popular editor of this category.

TECO

TECO, the Text Editor and COrrector, (c. 1970) is an interpreter for a string processing language. TECO can be used interactively as a stream-oriented editor; its basic commands can also be used as building blocks to provide quite elaborate editing operations. Many variations exist (DEC TECO and TENEX TECO are two), with varying capabilities and syntax. The conceptual model considers a document to be a sequence of characters, possibly broken into variable-length virtual pages by form-feed characters, and into virtual lines by line-end characters. Pages may be combined in an in-core editing buffer considered to be simply a varying-length string whose length may grow up to the in-core memory available.

The interface is based on typed input, typically consisting of single-character command syntax of the form:

[argument][single character command]

Commands can be combined to form sequences. Regardless of whether the user specifies a single command or combined command, TECO does not interpret the command string until the user presses the ESC key. In the ensuing examples, the terminating ESC is implied. The editing buffer is the amount of the file in memory. The viewing buffer on the document defaults to the null viewing buffer. The document is displayed only upon explicit command; the user can specify a viewing buffer of any size as explained below.

TECO maintains the current position as a value called **point** (symbolized by "."), which simply contains the number of characters in the buffer to the left of it. This pointer can be positioned absolutely (by a numeric value), relatively (by a positive or negative character or line displacement), or by pattern searches. For example, in TENEX TECO [BBN73], `0J` or `BJ` jumps the pointer to the top of the buffer, `ZJ` jumps the pointer to the end of the buffer, `43J` jumps to the 43rd character of the buffer, `.-9` or `-9C` moves the pointer backwards nine characters, and `17;BJ` jumps to the top of page 17. The symbols `Z`, `B` and `.` are not simply command modifiers but are registers that contain the point for the end of the buffer, the point for the beginning of the buffer, and the current point respectively; thus, the above commands using these registers resolve to an absolute character address.

Though TECO is character-oriented, special commands allow the user to edit a document in terms of a line model. Again, using appropriate register values, L moves the pointer to the beginning of the next line, -L moves the pointer to the beginning of the previous line, 0L moves the pointer to the beginning of the current line. Similarly, :L moves to the end of the current line while -:L moves to the end of the previous line. Line-oriented printing commands are provided as well; 7T prints the characters from the pointer until the beginning of the seventh line after the pointer, T prints the segment of the current line after the pointer. We stress that lines are an abstraction provided to the user; the text is stored not as lines, but simply as sequences of characters that are interpreted as lines by a filter that understands special line-end delimiters. A more complicated filter, for instance, might be able to extract programming language constructs from the stream.

Fundamental commands such as insert, delete and context search are supplied. `INow is the time(CTRL-D)` inserts the string "Now is the time" before the current pointer. 5D deletes the five characters after the pointer. `Sgood(CTRL-D)` finds the first match for "good" after the current pointer and moves the pointer there; similarly, `Rgood(CTRL-D)bad(CTRL-D)` replaces the first occurrence of "good" with "bad."

Importantly, TECO also supports commands for conditional execution to aid in creating more complex commands. `Q-registers` are available for holding any numeric or string value. Simple uses include performing arithmetic and moving or copying strings. To move a string of text, for example, the string is first saved in a Q-register and then deleted from the buffer (in some versions, the deletion is automatic). Next, the character pointer is moved to a new location and the contents of the Q-register are copied into the buffer at this new point.

If a Q-register contains text, the text may be interpreted as a command string. Thus, TECO can be used as a programming language to build editing commands. Higher-level commands are created by joining together many lower-level operations. Consider the pseudo-code for a global change operation with query and replace prompting:

```
WHILE (pattern is found in source)
   IF user response = "Y" THEN
      substitute newstring for pattern
END
```

With this pseudo-code in mind, to query and replace "good" with "bad," one could write the TECO code:

```
J<Sgood(CTRL-D); V↑T-↑↑Y"E-4CRgood(CTRL-D)bad(CTRL-D)'>
```

J puts the pointer at the beginning of the buffer. The < > pair are loop delimiters, indicating that the commands inside the loop should be executed repeatedly. `Sgood(CTRL-D)` is the search command we have seen previously. Upon failure of the search, ; V skips to the end of the loop construct. The ↑T is a "variable" that is assigned the value of a character typed by the user, while the ↑↑Y is the value of a capital Y. The subtraction expression, ↑T-↑↑Y, equals zero only when a Y is typed in. Thus, if the preceding expression is equal to zero, then the commands following the "E are run; otherwise everything until a delimiting ' is skipped. The -4C moves the pointer to just before the beginning of good. Finally, the `Rgood(CTRL-D)bad(CTRL-D)` performs the appropriate replacement. The loop then repeats until failure.

The raw power of TECO is evident from the above example. The abstraction of text (a continuous stream of text with a pointer) is simple, especially for the programmer, as it parallels the abstraction of computer memory with associated program counter. Continuing this analogy,

TECO is to a text stream what assembly language is to sequential computer memory. The TECO language provides a powerful base for a trained systems programmer or for a compiler's code generator; however, it does not provide a reasonable high-level interface for the average user, just as assembly language does not provide a reasonable interface for the casual (and even proficient) programmer. The syntax is cryptic. While all commands operate at the point, user misconceptions of the exact point location often results in off-by-one errors. TECO has been used effectively as an implementation language in several editors, most notably in EMACS described below. However, we believe that it is not a proper tool for either knowledge workers or competent programmers because of its low-level orientation.

4.3 Display Editors

This category includes several editors based on work done by Deutsch [DEUT67] and on the work of Irons and Djorup [IRON72], as well as several editors with an Irons-like model. The simple Irons outline for a CRT editing system has been the backbone of many editors: NED [BILO77, KELL77], *bb* [REIS81], PEN [BARA81], Z [WOOD81] and sds [FRAS81]. We present a general overview of the standard functions available in this kind of editor and then describe in more detail the unique features of several specific instances.[12]

In the Irons conceptual model, text is conceived of as a quarter-plane extending indefinitely in width and length, with the topmost, leftmost character the origin of the file. The user travels through this plane by using cursor keys and changes characters by overtyping. At any time, the user sees an accurate portrayal of the portion of the file displayed. Text is input on the screen at the position of the cursor. The environment is "modeless"; since all typing on the screen is considered text, commands must be entered either through function keys, control characters, and escape sequences, or by moving the cursor to and typing in a special command line at the bottom of the screen.

The command syntax is typically single-operand postfix. Basic traveling and editing primitives are provided, such as `+/- pages`, `+/- lines`, `+/- words`, `insert character`, `delete word`, and `back word`. Some of these may be preceded by an optional modifier. Thus, `+ page` scrolls forward to the next page, while `3 + page` scrolls three pages. Additionally, the editing and viewing buffers can be moved left and right and multiple windows support easy interfile editing. These editors make use of pick and delete buffers; hence deleted text is not discarded but is put in a buffer for possible subsequent use for moving or copying text. Functions such as `delete`, `pick`, and `put` may be combined with element modifiers such as `character`, `word`, `line`, and `paragraph` to allow more familiar specification of deletes, copies, and moves. A marking facility allows the user to select with the cursor two arbitrary points in the text to define a scope not easily specified with the element modifiers.

For display, most of the Irons derivatives use special algorithms to minimize the amount of screen updating necessary.

Brown's *bb*

Brown's *bb* [REIS81] is a typical example of the Irons model editor. Running under the UNIX operating system on a VAX 11/780, it makes use of a wide range of function keys for interaction.

One of *bb*'s extensions of the model is the maintenance of an up-to-date temporary file on disk along with a linked list of changes that have been made to the old file. This change history

[12] In this general overview, the syntax used does not reflect that of any given system, nor does an example of a general operation imply that each system contains that operation.

serves as the backbone of the undo command, which is capable of reverting changes back to the beginning of the editing session.

For travel, as well as providing the standard +/- keys, *bb* allows the user to save positions in named buffers and to jump to these positions with a **goto** command.

bb provides user-manipulation of the instructions with the **do** facility. Rather than providing a macro language, **do** provides a mechanism for capturing and naming a group of keystrokes. In general, a **programming-by-example** facility is an extremely elegant, powerful tool for both the novice and the experienced user. The user does not have to think in terms of a macro language syntax (with associated variables, flow-of-control constructs, and textual verbosity), but defines the new operations in terms of the same syntax that is used for editing. Complex operations that are hard to specify in a a procedural macro are almost trivial in terms of keystroke macros where the user simply executes the commands while the system captures them. For example, to find all instances of a *troff/me* italic formatting command, a separate line of the form:

```
.i "this will be italicized"
```

and change them to the TEX form:

```
{\sl this will be italicized}
```

one could use the following keystroke macro (all capitalized words are commands implemented as function keys or control sequences):

`TOF`	[goes to top of file]
`DOBEG`	[begins capturing keystrokes]
`ENTER ^.i +REG-EXPR-SEARCH`	[search forward for an occurrence of ".i" which starts on a newline]
`{\sl`	[type over existing ".i "]
`INSERT-SPACE`	[inserts needed space]
`+EOL`	[goes to end of line, 1 char past the quote]
`BACKSPACE`	[put cursor at end quote]
`}`	[types right bracket over quote]
`DOEND`	[finishes capturing keystrokes]

Now every time the user presses the `DO` key, *bb* will perform all the keystrokes entered between the `DOBEG` and `DOEND` keys. *bb* does not support parameterized keystroke macros or macros that prompt for particular input and subsequently continue executing; hence, one could not design a general-purpose keystroke macro similar to the special-purpose one above.

bb examines the file extension (file type) of the current file and loads an internal table with target-dependent information. This allows *bb* to perform automatic indenting for various programming languages and to recognize structural entities such as paragraphs in documents. Like many of the editors in this category, *bb* supports multiple viewing buffers and windows, though it only maintains a single editing buffer.

bb allows users to bind their own personal keyboards to the standard commands by modifying a control file. *bb* also supports an invocation-time profile, allowing personalized defaults on startup. This is coupled with a state-save facility that maintains necessary parameters from session to session. A **help** facility allows easy access to a complete online manual. Screen manipulation is performed by looking up terminal capabilities in the UNIX *termcap* database [JOY81] to determine output device characteristics, and by using specialized screen-optimization algorithms.

Yale's Z Editor

Yale's Z editor [WOOD81] extends the general Irons functionality by providing facilities that aid in program creation while maintaining the general-purpose functionality of the editor.

Editor commands are entered using control characters coupled with **cursor** keys. Function keys are not used; the developers dislike the fact that the user's hand must be moved from the typewriter keyboard to use them. Software allows overloading of the standard ASCII character set by using certain keys as shift keys. The interaction language also supports the overloading of each editor command. Here, as in most of the Irons derivatives, one command may be made to do slightly different things by prefacing it with optional arguments. For example,

arg *string* **fSearch**

searches forward for the next instance of the pattern *string* and moves the cursor there if successful. Each command may be prefixed with the special command **meta**, slightly altering the function of the command to which it is attached. For instance, **meta fSearch** causes case-insensitive searching.

For travel, Z remembers the last seven buffer positions, allowing the user to review previous contexts while the current one retains the status quo. Like *bb*, Z allows the user to put a "bookmark" on a certain spot in the file for later return.

The unique features of Z are its solutions to the program-editing task. Rather than using the structure-oriented approach, in which the editor has specific knowledge of the syntax (and possibly the semantics) of a target programming language, the Z editor represents programs as text, offering visual cues and a tight interface with existing compilers and debuggers to take the place of the innate knowledge of syntax-directed editors.

The designers of Z contend that "existing structure-oriented program editors have several disadvantages, such as increased complexity in the implementation, a restrictive user interface, and poor support for editing." Their solution is to represent the program as a text while equipping the editor with knowledge of program elements such as quoted strings, end-of-line delimiters, and matching tokens (such as **begin/end**) that signal indentation, as well as with indentation rules.

This representation allows Z to perform many functions normally associated with structure editors. Prettyprinting for block-structured languages is done by examining the last token on a line when the newline key is pressed. If that token is in a table that lists it as a token requiring subsequent indentation or exdentation relative to the previous line, the next line will be appropriately indented or exdented. This algorithm gives the desired result most of the time; a manual mode is offered to correct any mistakes made by the automatic mode. The matching token table also drives commands that close off the most-recently-begun matching unit, that find the end of the nearest unit if already closed, and that skip over the matching expressions as single units. This is particularly useful in LISP programming, where levels of parenthesization are hard to manipulate. Again, this facility does not require the editor to have syntactic knowledge of the target language, but simply to maintain a table of matching tokens.

Syntax-directed structure editors allow the user to manipulate syntactic units as single entities, as well as to view levels of syntactic detail. Z provides analogous features based on indentation level, which the designers claim work because "all the important information about the block structure of a program is contained in the indentation, provided the programmer is consistent."

The designers of Z chose to do neither the syntactic checking nor the incremental compilation often associated with syntax-directed editors, as the philosophy is *not* to integrate the

compiler directly into the editor. Feeling that "the programmer is the person best able to decide when his program is in a state ready for compilation" and that "existing compilers are perfectly able to locate errors" [WOOD81], the designers of Z attempted to enhance communications between editor and compiler. The user can execute the `compile` command from the editor; this communicates with an asynchronous process that formats the compiler request per the target language, puts the request on the processor queue, appends error messages in a special file, and returns a completion message to the editor when done. The user can continue editing while this is being done.

Z also provides a link to `Multiple User Forks`, a program that maintains multiple user contexts in parallel. This allows the user to exit from Z into any of the other forks (perhaps to read documentation or check on the state of some running program) and to return to Z without loss of state.

EMACS

MIT's EMACS [STAL80, STAL81] is an "extensible, customizable, and self-documenting" display editor. Several versions and dialects exist, most notably the Stallman version for the Tops-20 operating system (from which our examples are derived), the Honeywell Multics version [GREE80], and the UNIX version by James Gosling of Carnegie-Mellon. Some versions of this full-screen editor for timesharing systems are written in TECO, others in the high-level language LISP. To extend or customize the functionality, users write routines in the same language as that in which the standard editor functions are written, rather than using an editor macro language. Richard Stallman, the designer of the first EMACS, feels that this capability allows the user to transcend any limitations imposed by the editor's implementors. The basis of the successful EMACS strategy is that defining the extensions or changes in the source language "is the only method of extension which is practical to use"; it is unwise to maintain a "real" implementation language for the implementors and a "toy" one for the users. The user is able to bind many extensions or changes in a library, which can be loaded at invocation time. In fact, many of the core facilities that exist today were originally user-written extensions and were later adopted into the production system, thereby encouraging arbitrary growth rather than design. EMACS does offer a keystroke macro facility with prompts, so that non-programming users do have an alternative to a programming language at their disposal.

In EMACS, every typed character is considered a command. The keys for printing characters are bound, by default, to a command called `self insert` that causes that character to be inserted into the text at the cursor location. Generally, non-printing characters (control and escape sequences) invoke commands to modify the document. The editing language accepts single-character commands and finds the current binding between command key and function in a table. EMACS and other display editors offer a `quote` facility that allows characters typically used as commands to be inserted as characters into a document. A few of its many interesting features include a `query replace` facility, `transposition` functions for lines and words, and an automatic balanced parenthesis viewer.

The windowing facilities are extensive as well. The system supports multiple open files and hence multiple editing buffers[13] with associated viewing buffers and windows. The `CTRL-X 2` command divides the screen into two windows containing the viewing buffers designated by the user. The cursor is in one window at a time; `CTRL-X O` switches between windows. Special "narrowing" commands serve to change the size of the viewing buffer and editing buffer while

[13]Note that our use of the term "editing buffer" here, while consistent with our previous usage, differs slightly from Stallman's terminology in describing EMACS [STAL80, STAL81].

leaving the window intact. The user marks one end of a region, moves the cursor to the other end and issues the command CTRL-X N. This range now defines the maximum range of the editing buffer. If it is larger than a window's worth of text, the viewing buffer is set to a full window's size; otherwise it is set to the size of the editing buffer.

The fact that all keys (including alphanumerics) are bound to actions is very important, as the **self insert** action can be extended to effect more complex results. For example, one can extend the definition of the space character to insert itself and to check to see if an automatic wordwrap is necessary. A more detailed use of the key redefinition facility is the EMACS abbreviation package. Here, all the punctuation characters are redefined to look at the previous word, to check for its existence in an abbreviation table, and if it exists, to substitute the expanded word for the abbreviation as the user types.

EMACS offers *major modes*, editing environments tailored for editing a particular kind of file. For example, *text mode* treats the hyphen as a word separator; LISP mode distinguishes between lists and s-expressions. The major mode can automatically define different key bindings for a particular application. For example, in many programming language major modes, the tab key is redefined to provide automatic indentation.

EMACS can keep a record of all keystrokes typed in a session in a **journal file**. If a system crash destroys a current editing session, the user can instruct EMACS to bring up an old version of the file and replay the keystrokes from the journal file. The user watches the changes being made, and can stop the process at any time. (This allows a primitive undo facility: the user can replay up to a desired point and then discard the rest of the changes that are no longer wanted.)

Another interesting facility for program editing is the TAGS package. The separate program TAGS builds a TAGS table containing the filename and position in that file in which each application program function is defined. This table is loaded into EMACS; specifying the command **Meta** . *function-name* causes EMACS to select the appropriate file and go to the proper function definition within that file. Other special libraries incude DIRED, a subsystem for editing a file system directory using the full-screen display capabilities, and BABYL, a complete message-handling subsystem. INFO reads tree-structured documentation files, performing the necessary operations to travel from one node to the next.

EMACS has a very large and faithful following in the academic research community. While the basic editor is not vastly different in functionality from Irons model editors, the customized, application-specific packages have "sold" the system. To obtain a distribution of EMACS, one must agree to redistribute all extensions that one develops. By now, these extensions are quite numerous and powerful. Thus, it is not the raw editor, but the editor and its extensions that far exceed the capabilities of most other editors. While the programming language certainly cannot be used competently by the average user, the availability of extensibility features for programmers has manifested itself in many powerful facilities.

Extensibility, however, has negative points. Though the major new packages are distributed to all EMACS installations, many customizations are personal ones. This leads to situations in which two people using EMACS have different syntax and functionality. One set of keyboard bindings might be different from the next (e.g., one CTRL-D moves down 1 line while another deletes a line) or, alternatively, an identical keyboard arrangement and apparently identical functionality may have fine distinctions that confuse a user from a different microcosm trying to use someone else's EMACS (e.g., one GOTO END-OF-WORD command might go to the last character in a word, preceding punctuation, while another may go to the first white space following the word). Thus, with extensibility in any editor comes the price of widespread divergence over

various installations and even over the same installation. The tradeoff between a large number of divergent but customized dialects and a single, standard language is unclear.

IBM's XEDIT

XEDIT [IBM80] is IBM's screen editor for their VM/CMS timesharing system. Unlike the Irons model editors just described, XEDIT uses local terminal intelligence to perform screen editing operations. The high-level conceptual model is the unbounded quarter-plane of text in which the user views a rectangular region, yet XEDIT still relies on the sequential card model in some of its operations.

The display editing functions of XEDIT only work on IBM 3270 or 3270-compatible terminals. These terminals have a local screen buffer memory and a special keyboard with keys that support editing on the local buffer, such as **add char**, **delete char**, and **delete to end of line**.

The user has several methods of command specification. The first is changing the screen image by driving a cursor and using the local editing capabilities. The user is able to edit both the displayed text and a status line that displays file name, record length, and several other options. The user changes the text and the status line by simply inserting, deleting, or changing the options field of each line. Pressing the ENTER key causes the contents of the editing buffer to be sent to the host computer, which determines the difference between the terminal's local editing buffer and the host's internal data structure, and updates its internal data accordingly. This synchronization between the screen buffer and the internal data structure is an important component of an editor using a terminal with local intelligence. Note that throughout this editing process, the host processor is not signalled of any changes, regardless of how long the user has been editing a screenful of text, until the user presses the ENTER key to transmit that screen.

Besides the display-editor-style commands, XEDIT accepts typed commands that are almost identical to those of the CMS line editor; in fact on non-3270 series terminals, XEDIT operates essentially like the CMS line editor. These commands, typed in on a special command line, control those operations that cannot be done using the local editing buffer, including control functions, such as reorganizing viewing buffer-window mappings and ending a session, search commands, and some types of insert, move, and copy commands. Any of these commands can be bound to the 10 function keys on the keyboard. XEDIT cannot support selection by marking with a cursor because the current position of the cursor cannot be read by the CPU. While it does allow textual specification of region commands, it also provides the user with a *prefix* field before each line on the screen (see Figure 13), to give additional functionality.

As we see in Figure 13a, the D on the line beginning "THE HIPPOPOTAMUS" marks this line for subsequent deletion. The **2a** is an instruction to add two blank lines after this line. The DD is a grouping marker that delimits the beginning and end of a region of text to be deleted (the two DDs need not be on the same screen of text). The single A again stands for adding a blank line. When the user presses ENTER, the screen buffer is transmitted, and the host computer interprets the prefix fields (as well as any local editing), updates the internal data structure appropriately, and redisplays the updated text, as shown in Figure 13b.

Figure 14 shows a similar move command using the prefix fields. The mm is a grouping marker, much like the DD above, to delimit the beginning and end of a multi-line region of text to be moved, and the f is a marker signaling the line after which the moved text should be inserted.

XEDIT's local editing style offers both advantages and disadvantages. The use of the local

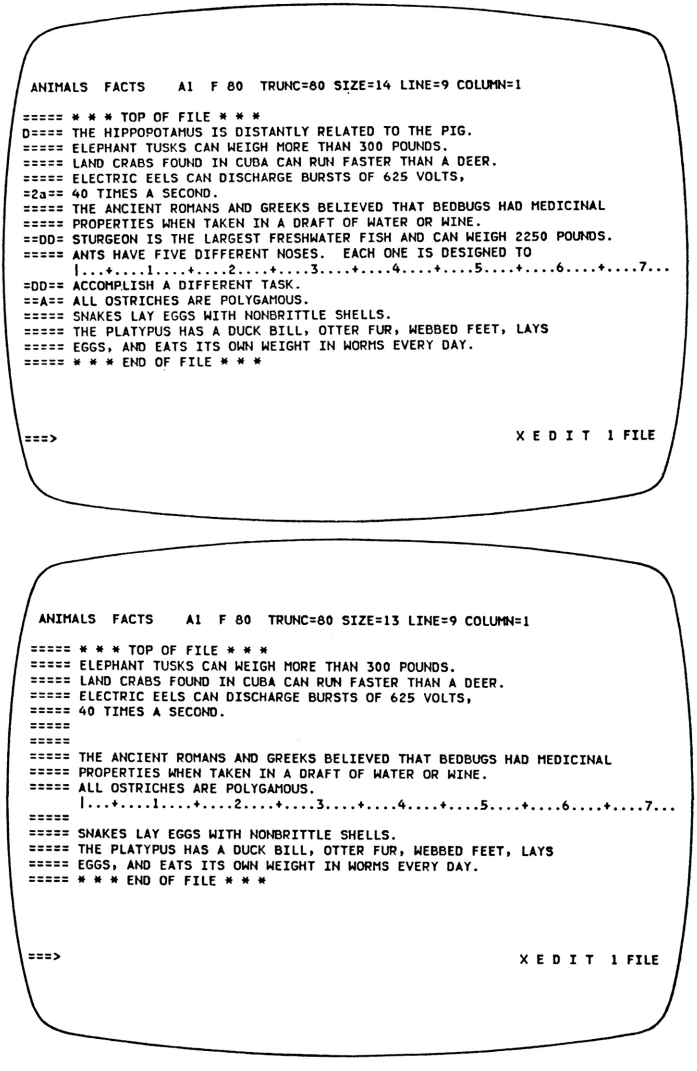


Figure 13. *Add and Delete with XEDIT prefix commands [IBM80].*

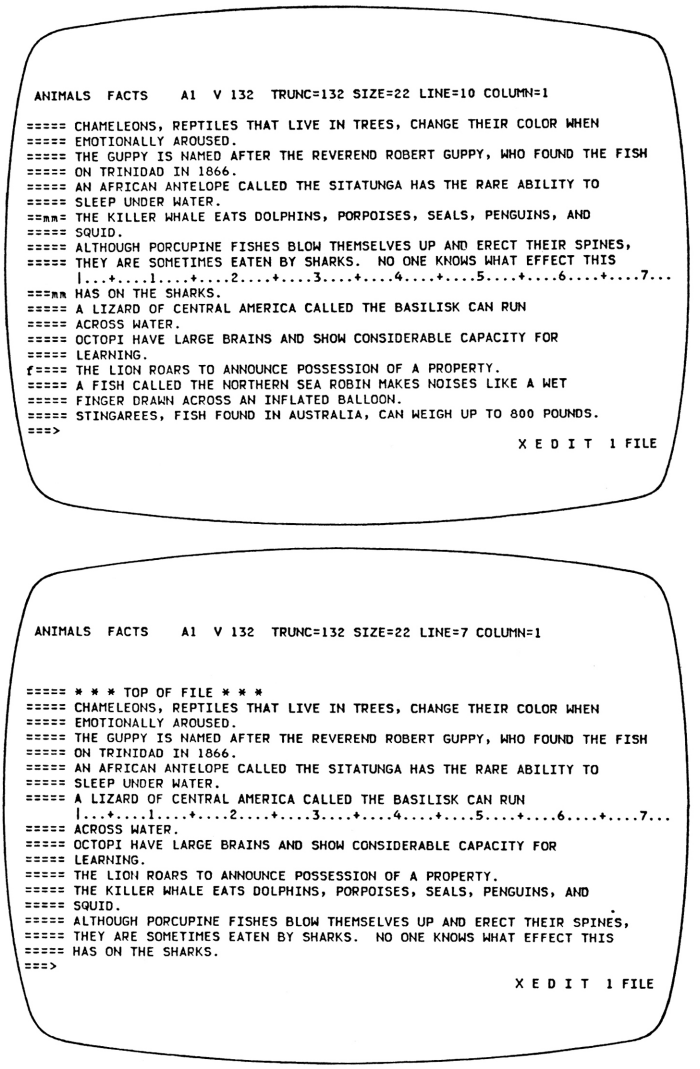


Figure 14. *Move with XEDIT prefix commands [IBM80].*

3270-series editing capabilities implies that users need not worry about an overloaded host system most of the time; most of the intraline editing, and even some of the block moving, as above, can be done without intervention of the host CPU. The editor is dependent upon the host system only when a screenful of text must be transmitted or a textual command (like search) must be executed. On the other hand, the local buffer offers no safety; if the host system crashes while a user is screen editing, all modifications on the local buffer are lost. More specific to XEDIT, the inability to do region selection within lines (because marking without CPU intervention is impossible on the 3270), reduces the generality of the editor. Additionally, the several styles of commands (typed, cursor-driven, prefix field) can confuse a novice user.

4.4 Graphics-Based Interactive Editor/Formatters

Xerox PARC's Bravo

Xerox PARC's Bravo (c. 1975) is one of the first of the interactive editor/formatters based on the display of high-resolution, proportionally-spaced text. Bravo allows the creation and revision of a document containing soft-typeset text with justification performed instantly by the system. The conceptual model is of a continuous scroll of typeset text that can be paginated when desired.

Bravo runs on Xerox's Alto, a 16-bit minicomputer with a raster graphics "portrait" display (roughly 8-1/2-x-11 aspect ratio) of 606-x-808-pixel resolution. This high-resolution pixel-addressable display allows more complex visual cues (overlapping windows, typeset facsimile text, graphics) than does the normal CRT terminal. A mouse drives a cursor and offers three buttons (called `left`, `middle` and `right`) that can be read independent of the cursor.

Bravo offers a mix of graphical and keyboard user interfaces. By moving the mouse, the user drags the cursor across the screen. The cursor addresses characters, special "menu" items, and other selectable elements on the screen. The interaction language is modal: the user can be in either **command mode**, in which text elements can be selected and commands initiated, or **typing mode**, in which keyboard text is entered into the document. Because of the modes, the user can specify commands with single alphanumeric characters; unlike many display editors, alphanumeric characters are not entered into the document unless the user is in typing mode. Commands not invoked with single alphanumeric keys are invoked with control characters.

As shown in Figure 15, the Bravo screen is divided into several areas. The **system window** contains information concerning what the user has just done and what can presently be done. The **document window** contains a viewing-buffer's worth of the document text scroll. The **line bar** and **scroll bar** are graphical entities that help the user travel through the document.

To travel in Bravo, the mouse is used to move a double-headed arrow cursor along the scroll bar, a vertical strip at the left side of the document. Pressing the `left` button on the mouse while the arrow is pointing to a line in the document's window causes that line to become the top line in the window; pressing the `right` button causes the top line in the window to move to the line the cursor is at. For more extensive traveling, one is supplied with a graphical **thumbnail** that moves along the scroll bar, and a **bookmark** that indicates on a graphically-displayed linear continuum from "front cover" to "back cover" the proportional current position in the document. If the user is half-way through the document, for instance, the bookmark indicates a point half-way across the continuum. To travel to a part of the document preceding what is being viewed, the user simply places the thumbnail somewhere before the bookmark on the continuum and presses the `middle` button; the document will "fall open" at the corresponding position in the text. By placing the thumbnail after the bookmark on the continuum, the user can similarly travel through the remainder of the document.

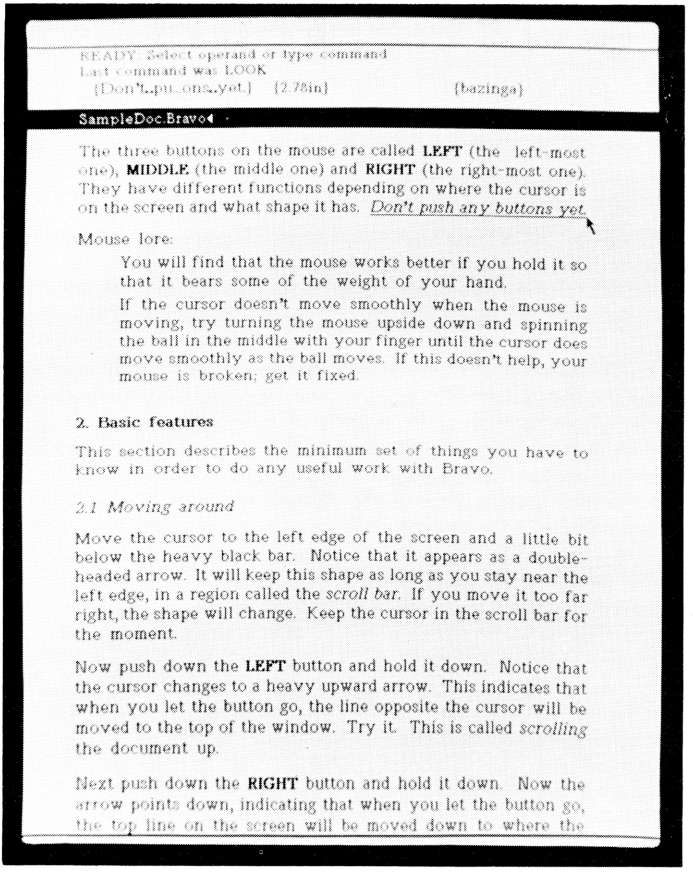


Figure 15. *Bravo Display*

Bravo uses a postfix/infix interaction syntax: a selection is followed by the **command**, followed by an optional **argument**. For example, deletion works by selecting the scope and pressing the D key, while insertion works by selecting the scope, pressing the I key, typing the desired text, and finally pressing the ESC key. Unlike the Irons model editors, Bravo is modal: characters are only inserted into the document during special modes, and are interpreted as commands in others.

Selection operates on four main elements: characters, words, lines, and paragraphs. The **left** and **middle** buttons of the mouse are used to select items, while the **right** button is used to extend those selections. With the cursor in the text area, the **left** button would cause the addressed character to be selected as the scope, while the **middle** button would cause the addressed word to be selected. To select the large elements, the user moves the cursor into the line bar. In the line bar, the **left** button selects a line and the **middle** button selects a paragraph. Extending the selection allows the user to specify a scope that lies between two of the entities addressed. Thus, clicking **left** in the text area would cause a single character to be selected; clicking **right** at some other character would cause all the text between (and including) the two selected characters to be selected. Similarly, a **middle right** sequence would select all the text between and including two words. In the line bar, a **left right** sequence selects all the text between and including two lines; a **middle right** sequence selects all the text between and including two paragraphs.

Operations are typically performed on the current selection. To delete a word, the user simply selects a word by clicking the **middle** button and then types D to execute the delete command. Similarly, to delete all the text between and including two paragraphs, the user clicks **middle right** in the line bar and types D. Changes are done analogously. To replace a word, the user clicks **middle** and types R. Bravo deletes the selected word and puts the user into insert mode; everything the user types until the **escape** key is pressed is inserted in place of the old word. The **Append** and **Insert** commands allow the user to add text in a similar manner without first deleting a selection. Bravo supplies an undo facility that undoes only the last operation.

Files are never saved until the user explicitly saves them. However, Bravo keeps a transcript of the operations that have occurred in the editing session. One can run BravoBug with the transcript against the old version of the file to interactively replay the editing session. The user is given the choice of single-stepping through each change or running the entire transcript, stopping whenever desired.

At the time of its introduction, the most innovative features of Bravo were its interactive formatting facilities. Bravo's unit for specifying the formatting attributes of text is the **look**. Each character in the document has associated with it particular looks; the looks of any character can be displayed by selecting that character and typing L ?. The looks specify a large assortment of type attributes: font style, point size, subscripting, superscripting, centering, justification, nested indenting, and leading (interline spacing), to name a few, are attributes that the user can change by typing L followed by a one-character operand. Other look attributes cannot be changed directly by command but are constrained by previous formatting attributes. As soon as a look command is executed, the document is dynamically reformatted to effect the revision – the document is up-to-date in both format *and* content at all times. A special page-format mode allows the user to see the document paginated as it will be printed.

Bravo does not allow integrated graphics, but provides output that can be postprocessed to add figures from the PARC interactive picture-editing systems [BAUD78, NEWM78, BOWM81].

Xerox Star

Star [SEYJ81, XERO82, SMIT82], Xerox's commercial successor to the Bravo, is, in terms of its user interface, the most advanced commercial product for office automation on the market at the time of this writing.

Like the Alto, the 8010 workstation on which Star runs is a personal computer with access to shared resources such as file- and printer-servers via an Ethernet network. It has a "landscape"

13-1/2-x-10-1/2 inch screen with a resolution of 1024 x 809 pixels, capable of displaying both a full page of a document and a large menu area.

Several design goals are important to the understanding of Star's interface and functionality:

- The designers determined that users should simply point to specify the task they want to invoke, rather than remember commands and type key sequences. They believed that the user should not need to remember anything (of consequence) to use the system.
- An important consideration was the development of an orthogonal set of commands across all user domains: the copy command in the text formatter, for example, should have similar semantics to one in the statistical graphing package.
- The system was designed to operate by "progressive disclosure." Star strives to present the user with only those command choices that are reasonable at any given juncture.
- Finally, Star is an interactive editor/typesetter; the screen is, for the most part, a facsimile of what the final document will look like.

The Star development team, which considered possible models for several years, remarks [SMIT82]:

> The designer of a computer system can choose to pursue familiar analogies and metaphors or to introduce entirely new functions requiring new approaches. Each option has advantages and disadvantages. We decided to create electronic counterparts to the physical objects in an office: paper, folders, file cabinets, mail boxes, and so on – an electronic metaphor for the office. We hoped this would make the electronic "world" seem more familiar, less alien, and require less training. (Our initial experiences with users have confirmed this.) We further decided to make the electronic analogues be concrete objects. Documents would be more than file names on a disk; they would also be represented by pictures on the display screen.

The high-level conceptual model of the environment is that of a desktop in which multiple documents can be operated upon simultaneously. Star uses a two-button mouse and a postfix interaction syntax. Rather than presenting the user with a simple textual menu or a list of available options and files, Star presents the user with graphical icons that resemble the entity to which the user is referring (see Figure 16).

To open a file for editing, the user simply points to the iconic file drawer that symbolically holds the document (noun selection) and issues the **open** command (verb selection). Choosing **open** causes a file drawer directory, containing identifiers for file folders and individual documents, to fill part of the screen. The user can open, copy, move or delete any of these folders or documents; touching **copy**, for example, causes a new document icon to be placed on the user's "desktop" area on the screen. Selecting this icon and **opening** it causes the editor to open a window that is large enough to hold a facsimile of an 8-1/2-x-11 inch page (see Figure 17). Editing operations similar to those provided by Bravo can be performed in this window. The interface, however, does not use control characters. Mouse buttons and function keys provide the most frequently used commands; a menu of window-specific commands appears in the window banner at the top of the window for selecting with the cursor, and a menu of infrequently used system commands is always available by selecting a menu icon in the upper right corner of the screen.

Traveling buttons located on the bottom and right borders of each window, as shown in Figure 17, are selected with the mouse. The ⊢ on the bottom makes sure that the left margin

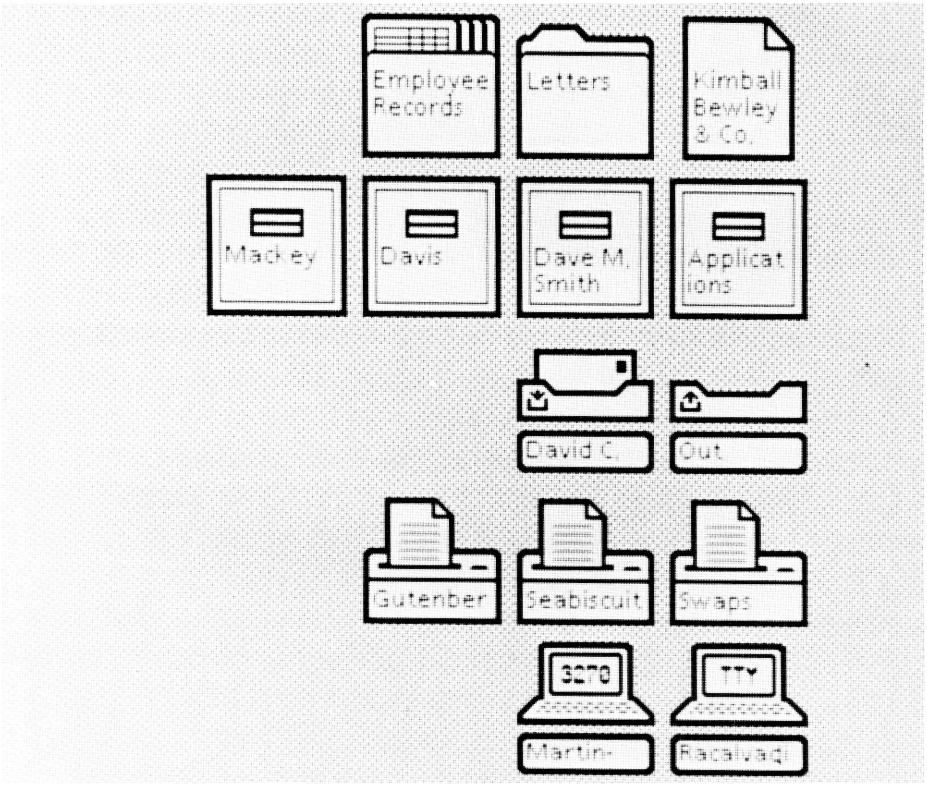


Figure 16. *Star Icons*

of the document is in view while the ⊣ makes sure the right half of the document is in view. The → scrolls the document to the right, the ← scrolls the document to the left, the ↓ scrolls the document downwards, and the ↑ scrolls the document upwards. `P` goes to the previous document page, while `N` goes to the next document page.

Other icons include a printer icon, a floppy disk icon, and an in/out box icon. To print a file one simply selects the appropriate document icon and places it on top of the printer icon. The programmable cursor changes to an hourglass to indicate that processing is taking place. Similarly, electronic mail is sent by placing a document on the **out box** and is received by selecting the **in box**.

As in Bravo, the mouse is used to drive the cursor and select elements. Selections are performed with the left mouse button and can be adjusted with the right mouse button. To select a character in the text, the user clicks the left button. Subsequently, when the right mouse button is held down, all the characters between the selected point and the current position of the cursor will be highlighted in reverse video; when the right button is released the highlighted area becomes the selection. Two left button clicks select the word containing the cursor; holding down the right mouse button extends the selection to include full words between the selection points and the cursor. Three left button clicks select the sentence containing the cursor; holding down the right mouse button extends this selection by sentences. Four left mouse button clicks select the paragraph containing the cursor; holding down the right mouse button extends the selection by paragraphs. A fifth left button click returns to the original character selection.

Most commands are postfix, requiring simply the selection of an icon or a region of text

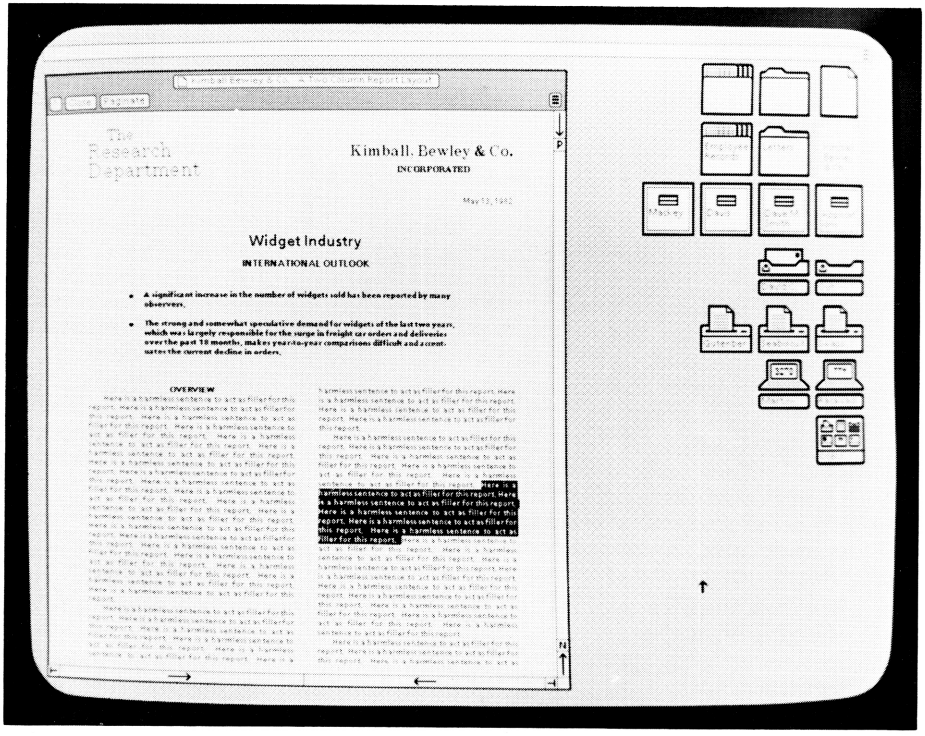


Figure 17. *Star Document in Window*

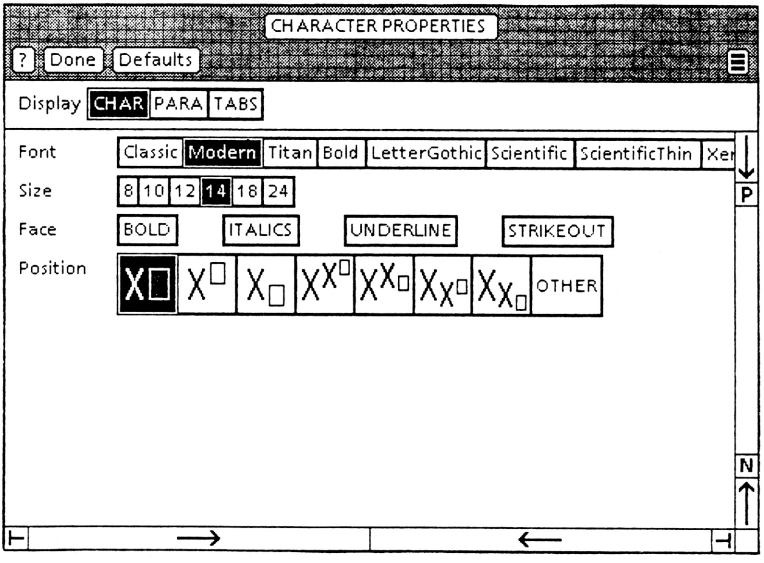


Figure 19. *Character Property Sheets*

followed by the issuance of a command using a function key or menu selection. Commands such as **find**, **move** and **copy** that need multiple operands are specified in infix or prefix, as appropriate. To perform a **find**, the user presses the **find** function key and is given a **find** property sheet to fill out, as shown in Figure 18a. The user fills in the **Search for** box by typing a search pattern and specifies other attributes of the search by selecting various options on the property sheet with the mouse. (Here **TEXT**, **IGNORE CASE**, and **ENTIRE DOCUMENT** are selected). The options that are selected remain selected from search to search until the user explicitly alters them. To perform the search, the user selects the **Start** button in the window menu. While Star is searching, it displays the message "**Searching**..." as feedback for the user. The **?** and **Cancel** button provide help and abort the search, respectively.

The search and replace operation uses the same property sheet. Picking the **CHANGE IT** button on the **find** property sheet brings into view a second set of properties. The user can now type in the pattern to **Change to** and specify what should be altered and whether the replacement should be done with confirmation. When performing these operations, the message "**Substituting**..." provides needed feedback.

Like Bravo, Star performs instant formatting and justification in the proper type size and style. Associated with each element (element being any entity from a character to the entire document itself) are property sheets that contain status information for that element (see Figure 19). Initially, attributes in property sheets have system-assigned default values. To change the typeface of a particular character, the user selects the character, presses the **props** key, points to the desired typeface in the property sheet and **closes** it. The change takes effect immediately.

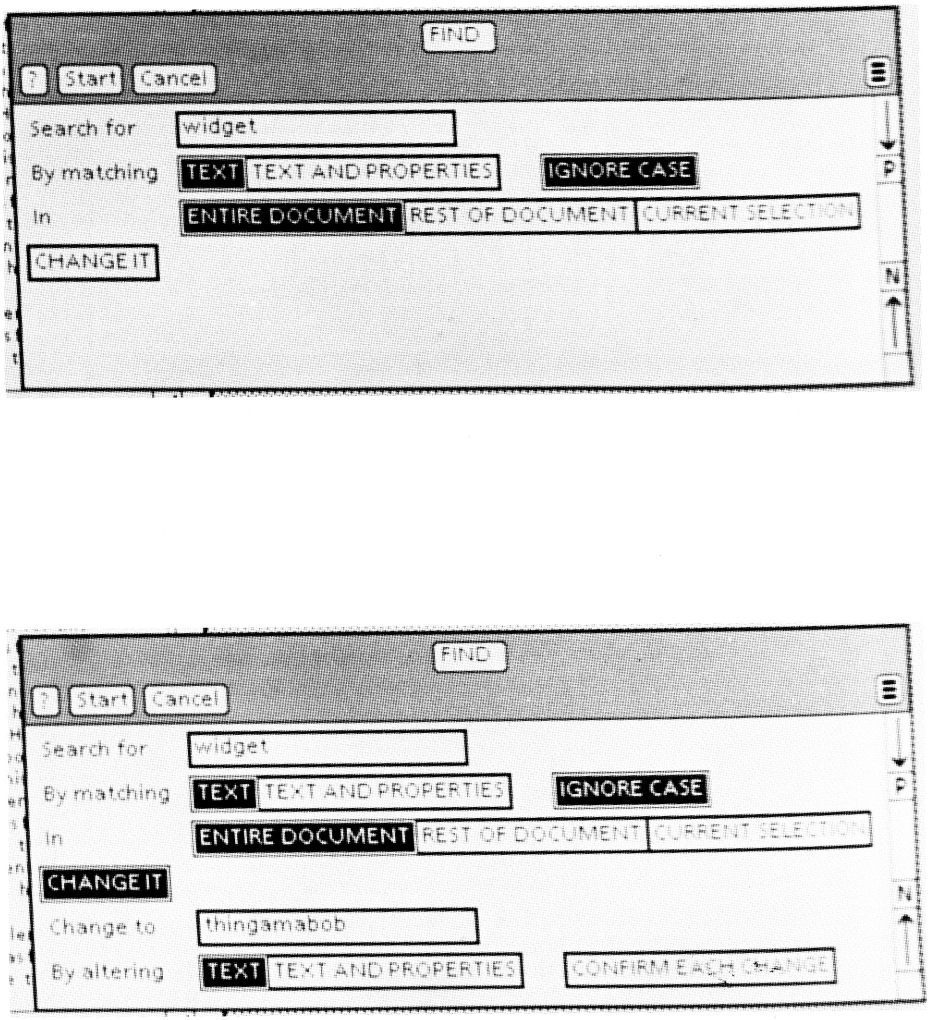


Figure 18. *Search, Search and Replace Operations*

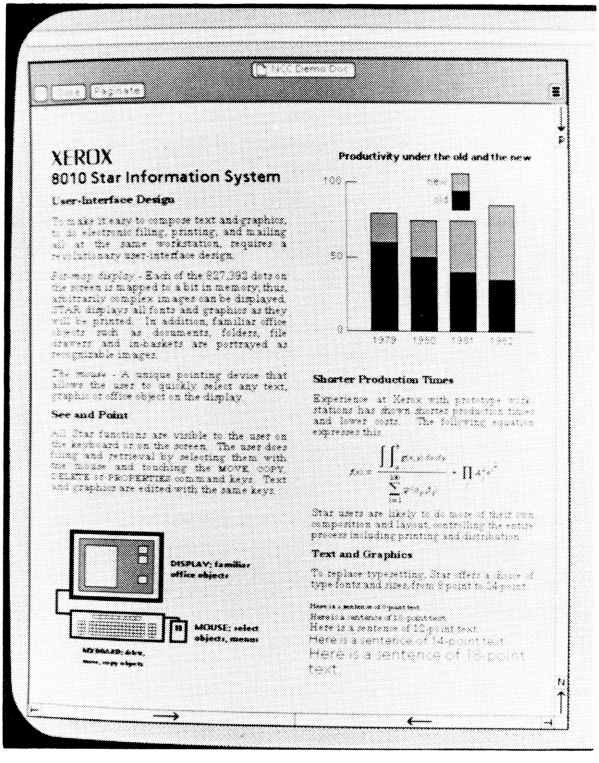


Figure 20. *Star Graphics*

(Note that the property sheet is presented in the same kind of window as a normal document. In fact, to see all the available typefaces in this example, the user would have to scroll the document to the left with one of the scroll symbols.) Star enables the user to define a standard collection of property sheets to provide document templates (style sheets), as in a database-driven formatter such as Scribe. The user simply copies the template and enters the new text, assured that the basic format is properly defined. The designers compare this to tearing off a standard form from a preprinted pad [SMIT82].

Star provides a drawing package, the results of which can be integrated into a document (see Figure 20). The user selects lines, boxes, shading patterns, and other primitives from a menu and uses these to draw on a user-determined grid. Just as selections can be extended, several graphic items can be selected at once by holding down the appropriate mouse buttons. Users are also allowed to define clusters of graphical items to form new "primitives." Graphics can be

scaled up or down to fit in a fixed space in a document. Star also provides packages for making and editing bar charts and spread sheets, and for retrieving information through a relational database system. The designers have stressed what we believe to be a vitally important concern: that all the packages have consistent interfaces, as users especially want a particular command to behave in a consistent way in the provided multi-view environment. Whether in the text editor, the graphics editor, or the chartmaker, the user issues commands by selecting the object of the operation and issuing the appropriate command through function keys or menu buttons. To delete a word, one selects the word and presses the **delete** key; to delete a rectangle, one selects the rectangle and presses the **delete** key.

Besides its carefully-crafted user interface, Star provides some interesting solutions to typical online manuscript-preparation problems. Mathematical and foreign-language typesetting in most systems involves using escape/control sequences or long English-language mnemonics to represent the special characters. Star presents the **virtual keyboard**, a graphical representation of the keyboard on the screen. To use a key, one simply points at it or presses the corresponding physical key. Star has knowledge of mathematical symbols and can construct complex equations and formulas as they are typed, changing the size of the symbols used as equations get larger or smaller. Editing continually adjusts the horizontal and vertical spacing and placement of subscripts and superscripts.

ETUDE

ETUDE [HAMM81] is a document production system designed with twin goals: "to *extend* the functionality of conventional word processing systems while *reducing* the complexity of the user interface." Unlike Bravo or Star, ETUDE uses prefix syntax:

action modifier element

where an *action* might be **move** or **delete**, a modifier might be a number or a word like **start-of** or **next**, and an element might be **paragraph**, **word**, **document**. The designers feel that the prefix syntax, as in "**delete 3 words**," more closely approximates natural language, and thus is preferable.

Like Bravo and Star, ETUDE is an interactive editor/formatter, providing typeset, formatted text on a standalone workstation with a bitmapped screen. ETUDE has adopted a Scribe-like method for describing formatting, switching the burden of complex formatting from the user to a document database that contains standard formats for a range of documents and document components. In a letter, for example, the ETUDE system will do special formatting for the **returnaddress**, **address**, **salutation**, **body**, and **signature**. While ETUDE always keeps the up-to-date formatted document on the screen, it uses the left margin of the screen as a **format window** to place formatting descriptor tags that indicate the type of high-level action that has been taken on a particular section of text (see Figure 21). This technique attempts to bridge the gap between the unformatted but explicitly expressed formatting code, and the displayed facsimile page that, once formatted, often does not contain information about the act that caused the formatting to occur. (A more detailed discussion of the interactive vs. batch formatting question is presented in Section 5).

The user interface is designed for various levels of expertise. The user can call a menu to the screen at any time and select a command with cursor keys or a pointing device. Alternatively, the user can type a command to perform the same action – or use specialized function keys provided for the most widely used commands.

The system provides a **cancel** command to abort the current operation, an **again** command to execute the current command again, and an indefinitely deep **undo** facility. The same tree

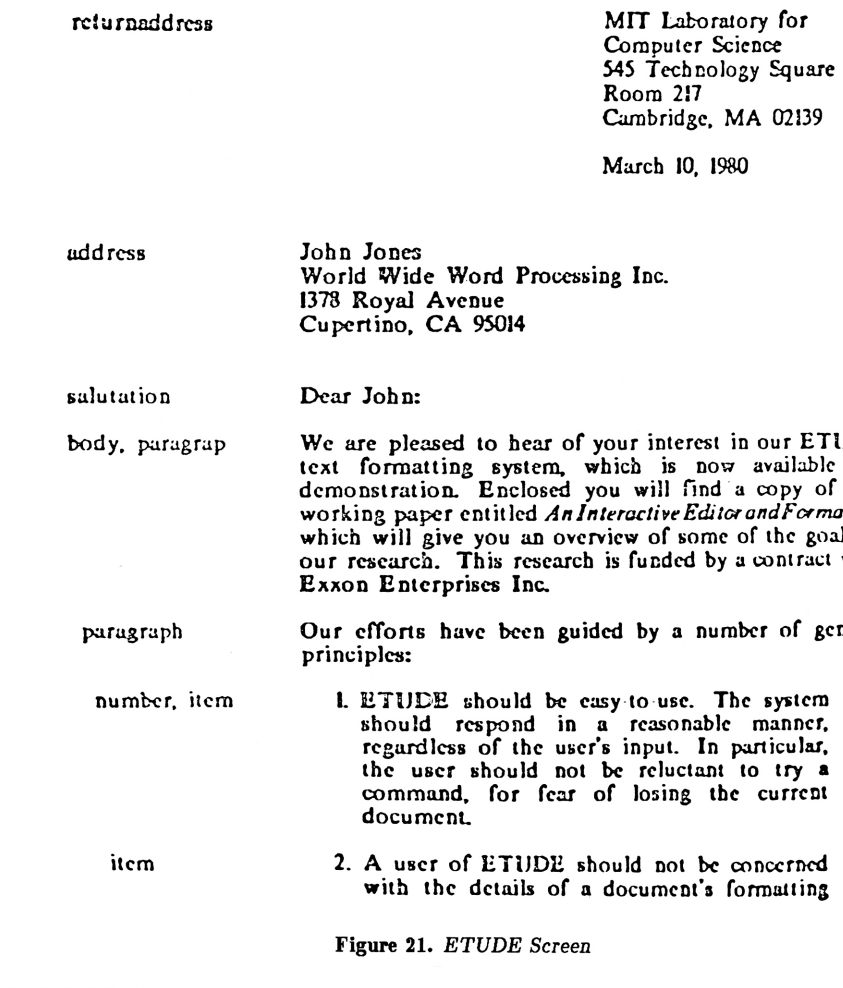


Figure 21. *ETUDE Screen*

structure that keeps track of the undo history is used in a `help` command that creates windows to show the user the session's history and what options are currently available. When the `help` command is invoked the user is presented with descriptions of a few past operations plus what is currently being done.

4.5 General-Purpose Structure Editors

Structure editing, pioneered by Englebart with NLS, has been "rediscovered" as an alterna-

Operation	Effect on Tree
INSERT	Insert a new site (with empty data collection) into a specified gap
DELETE	Delete a subtree (nodes and data)
COPY	Copy a subtree to a specified gap
MOVE	Move a subtree to a specified gap
SPLIT	Split a node and its data into two
MERGE	Merge two nodes and their data
EXPAND	Insert an intermediate level in the tree
SHRINK	Delete an intermediate level of the tree
ORDER	Permute the nodes on a tree level

Figure 22. *Tree Editor Functions of a Structure Editor [BURK80]*

tive to standard character-oriented methods of editing. Since most target applications have some innate structure (manuscripts, for example, are composed of chapters, sections, paragraphs), the philosophy of structure editors is to exploit this "natural" ordering to simplify editing. The most common representation is a hierarchy of elements. Standard operations on this tree structure, as taken from XS-1[BURK80], are shown in Figure 22.

NLS/AUGMENT

NLS was a product of research at the Stanford Research Institute (now renamed SRI, International) between the early 1960s and late 1970s. Renamed AUGMENT and marketed by Tymshare, Inc., NLS is one of the seminal efforts in the field of text editing and office automation; indeed, many of its features are being reexamined and reimplemented today – almost 20 years since the inception of the NLS project. For example, NLS introduced the notion of conceptual models for the editing and authoring processes, (tree-) structured editing, element modifiers for the editing and viewing operations, device-independent interaction syntax, the mouse as a cursor-manipulation device, sophisticated browsing and viewing mechanisms, intermixed text and graphics, and even multi-person, distributed editing. At a spectacular, landmark demonstration of the system at the 1968 Fall Joint Computer Conference in San Francisco, text, graphics, and live video of Doug Engelbart in San Francisco and colleagues twenty miles away in Menlo Park were superimposed on multiple viewports on the (video projected) screen, as they were working

together and explaining what they were doing. "Chalk-passing" protocols were demonstrated for synchronizing multiple users. This demo was a forerunner of graphics- and sound-based teleconferencing.

NLS/AUGMENT clearly embodies much more than just a text editor. Its aim is to provide a new way of thinking and working by utilizing the power of the computer in all aspects of one's work [ENGE68]:

> We are concentrating fully upon reaching the point where we can do all of our work on line – placing in computer store all of our specifications, plans, designs, programs, documentation, reports, memos, bibliography and reference notes, etc., and doing all of our scratch work, planning, designing, debugging, etc. and a good deal of our intercommunication, via consoles.

Regardless of the subject matter, all NLS information is stored in a hierarchical outline structure of the form:

```
1 ...
  1a ...
  1b ...
    1b1 ...
      1b1a ...
    1b2 ...
    1b3 ...
    1b4 ...
    1b5 ...
2 ...
  2a ...
3 ...
4 ...
  4a ...
    4a1 ...
    4a2 ...
```

Statements can be nested an arbitrary number of levels. Each statement has associated with it a statement number of the form shown above; these are the main means of referencing the statements from other parts of the text. One statement may be a **substatement** of another statement (1a1 is a substatement of 1a), one may be the **source** of another (1a is the source of 1a1), one may be the **predecessor** of another (4a1 is the predecessor of 4a2), or one may be the **successor** of another (4a2 is the successor of 4a1). NLS provides modifiers to reference not only text elements but structure elements as well. A **statement** is a text node of up to 2000 characters. A **branch** is a statement and all its substatements. A **plex** is a branch plus all the other branches with the same source. The plex of 4a1 is 4a1 and 4a2; the plex of 4a2 is the same. A **group** is a subset of a plex; it consists of all the branches of a plex that lie between and include two branches. The group of 1b2 and 1b4 includes 1b2,1b3, and 1b4.

The hierarchy is useful for programs as well as for documents since it can be used to model the block structure of the program. Viewspecs allow levels of detail in the outline structure to be made invisible; the viewspecs effect **information hiding**, the selective display or non-display of existing material based on attributes provided by the user.

NLS/AUGMENT allows the user to create a hypertext by superimposing on the structure a network of links that point to various discrete statements in this document. In general, these links are specified by the identifier

`<host,owner,file,statement>`

which allows linking of documents over multiple computers.

Commands in NLS/AUGMENT can be executed by using a mouse to select from a menu on the screen, by using the keyboard, or by using both the keyset (described in Part I, Section 2) with one hand to enter the command, and the mouse with the other to make a selection.

The editing commands are quite extensive, providing the first attempt at an orthogonal command syntax with element modifiers. For instance, the **insert** command can be modified, with nouns such as **word**, **sentence**, and **branch**. As in most structure editors, the commands are divided into those which operate on the structure (such as **move**) and those which operate on the text. NLS/AUGMENT provides a very large repertoire of both. Most standard tree manipulations are allowed, such as locating or deleting the next node or the previous one, locating the first subnode, and rearranging neighboring nodes. The **move** and **copy structure** commands provide dynamic renumbering of sections and updating of links throughout the document if necessary.

The system provides the ability to imbed control codes in special delimiters within the text both for formatting options such as font changes and for travelling information (links, annotations). These codes can be edited like regular text until they are invoked by special commands (a link is not operable until the jump command is invoked). Viewspec parameters allow one to turn off viewing of these special codes as desired.

A journaling facility provides extensive archiving power for past online conversations and teleconferences. TYMSHARE's commercial version of AUGMENT makes use of TYMNET, a transcontinental satellite network, to satisfy one of the original goals of the project: the sharing of knowledge across great distances. In fact, it is not uncommon for someone in New York to compose a document by making several links to an existing document belonging to a colleague in California.

At its time of introduction, NLS was unusual not only in terms of its functionality but also because of the software engineering environment in which it was produced. This environment included compiler-compilers, systems-implementation languages, and command language interpreters.

Burkhart/Nievergelt Structure Editor

Burkhart and Nievergelt at the Institute for Information in Zurich have designed a family of structure-oriented editors called XS-1 [BURK80]. The designers contend that the basic set of editing operations, regardless of the target being manipulated, are similar, and that "a universal structure defined on all data within a system" exploits that similarity to its greatest advantage. As in NLS, the structure of data of all types in XS-1 is represented as a tree, with the nodes ("sites") representing subsets of data. Like many structure editors of its kind, the core of the XS-1 system is a flexible tree editor that allows the user to manipulate the elements at the site (node) level. Fundamental to the XS-1 philosophy is the belief that the user works only on a restricted set of data and with a restricted set of commands at any one time. Therefore, the system supports progressive disclosure, explicity showing the user the valid command repertoire and operation targets at any given moment. The user always has the familiar tree operations available at all times.

XS-1 provides the user with standard structure editor methods of travel through the **explore** command. Here, the user can use relative motion to traverse up, down, left, or right in the tree. As well, absolute motion allows the user to move explicitly to something by specification of an identifier such as a name.

The tree editor follows several basic principles. After completion of any operation, the integrity of the tree structure is guaranteed. (This may be accomplished by attaching target-specific syntax rules to operations, making a syntax-directed editor.) XS-1 provides the ability to specify different views of the same targets, such as a tree structure of a program or an indented view of the same program.

An important aspect of XS-1 is the combination of the same target-independent tree editor with target-dependent back ends to create multiple editors. One such combination is a document editing/formatting system. Here, the author sees on the screen a rectangular window into the text and a text cursor. All high-level operations (**move**, **copy**, etc.) are handled by the target-independent tree editor; only a small set of text editing primitives, at the character, word, or sentence levels, are provided. The command set is consistent between targets; operations provided by the universal editor are also provided for specific, target-dependent modes, enabling the user to deal with a relatively small set of operations which do "obvious" things. For example, a **move** command in a text editor would move the selected text from source to destination while a **move** command in a graphics editor would move the selected graphics object from source to destination. Text formatting is done by appending a formatting descriptor to each site; these can be edited by the tree editor as well.

Fraser's s

Fraser's *s* [FRAS80] is an attempt to provide standard editing primitives which can be used to build a variety of editors. *s* allows the programmer quickly to create different front ends for a text editor so that various targets can be modified using existing editing routines.

The philosophy behind *s* is that many computer utilities – interactive debuggers, file system utilities, even tic-tac-toe games – are simply editors, in that they accept a particular input syntax and modify the existing representation and or state of their particular data. Rather than producing languages and scanners for each application, *s* attempts to use a generalized structure and a generalized text editor nucleus for editing all applications.

One application allows the user to edit UNIX **i-nodes**, complex (18-field) data structures containing pointers and information about a file block from the UNIX file system. When the system crashes or a disk block becomes unusable, the systems programmer occasionally has to go into the file system and manually change pointer values from a dump-type format. *s* provides a screen-based view of the file descriptor, allowing the user to edit each of the fields, which are represented one per line. An overstrike, for example, is translated into a call to the nucleus routine **fetch** to retrieve the appropriate field and a call to the nucleus routine **change**, to update the field. The deletion of a field would be performed with a call to the nucleus routine **delete**.

Another interesting use of *s* is as a UNIX file directory editor. The UNIX **ls -l** command provides a listing of file attributes:

```
drwxr-xr-x 2 nkm      224 May  9 15:27 bBACKUP
-rw-r--r-- 1 nkm    36585 May  2 16:42 section1.tex
-rwxrwxrwx 1 nkm    16714 Apr 25 17:11 section2.tex
-rw-r--r-- 1 nkm    48414 May  2 16:44 section3.tex
-rw-r--r-- 1 nkm    55282 May  6 00:23 section4a.tex
```

```
-rw-r--r-- 1 nkm    20113 May  6 00:49 section4a2.tex
-rw-r--r-- 1 nkm     9209 May  9 14:50 section4b.tex
-rw-r--r-- 1 nkm    22049 May  9 14:20 section4c.tex
-rw-r--r-- 1 nkm    26958 May  6 02:24 section4d.tex
-rw-rw---- 1 nkm     3362 May  9 15:10 section4e.tex
-rw-rw---- 1 nkm    18541 May  7 11:13 section5.tex
```

The first field contains a **d** if that entry is a directory; this field is not editable. The next nine fields contain **r**, **w**, **x** for read, write, and execute privileges for the owner, the group, and for all others, respectively, with a - indicating no access. The next field, the link count, is not editable. The next field contains the owner of the file. The rest of the fields are not editable, except for the last entry, the actual filename.

Rather than forcing the user to use the UNIX shell commands for performing renaming (**mv oldname newname**), deleting (**rm filename**), changing ownership (**chown filename**), and changing access rights (**chmod a+rwx filename** to allow all to read, write, and execute the file), the *s* directory editor allows the user to edit the listing directly, barring protected fields. Deleting the characters **r**, **w** or **x** removes read, write, and execute access for the corresponding parties; overstriking a - with **r**, **w** or **x** adds access. Typing over the owner name changes the owner, typing over the filename changes the filename. Deleting an entire line deletes that file.

A different front end allows the user to edit the state of a simple pedagogical computer. Rather than having the student submit punched cards in batch mode and easier and cheaper than having a physical laboratory machine, an *s* front end was written representing the machine data structure as editable lines and allowing the students to modify the appropriate fields. While the goals of the i-node and machine applications are different, the primitives to edit them, at least from a system view, are the same.

While the *s* editor was a limited experiment, its ramifications are wide-ranging. Many applications, especially ones that are computer-based, have some aspect which requires editing. We feel that Fraser's basic premise – that when changing a filename in a filesystem, when adding a user to a mailing list, or when editing a UNIX *i-node* as above, there is no reason why the user should have to resort to special maintenance programs – will be an important goal in the future of editing. As Fraser's *s* has shown, a general-purpose editor can be used to give the user a far more common interface across diverse applications than typically exists today. Moreover, with an appropriate interface, one can perform editing on a graphical representation of the target rather than on an unfamiliar, textual representation.

Walker's Document Editor

Walker's Document Editor is an attempt to design an editor for the preparation of complex documents such as technical manuals. An initial goal of the system [WALK81b] was to "develop a structured description for documents...distinct from any particular commands in the document source." The Document Editor uses EMACS as a base text editor and Scribe as a document-description database and compiler.

The Document Editor operates on a "document" as a collection of files in Scribe manuscript file form; it infers the structure of the document from the tags in the file being edited. The specialized functions for technical writing provided by the Document Editor are actually extensions to EMACS in the form of a user library.

The Document Editor provides four major categories of document structure editing commands: **locators, selectors, mutators**, and **constructors**. Locator commands allow the user to specify places in the document; these include commands to go up and down a structural level

(e.g., from section to subsection), to go to the next or previous item at the same structural level, and to go the next structural element of any type. Selector commands allow the user to determine the current makeup of the document by checking the status of the parts and the structure of the document at various (user-specified) levels. Mutators revise the structural makeup of the document, providing functions such as `change structural level` (make a chapter a section). Constructors allow the user to create and copy structural elements.

The Document Editor uses Scribe's commands for cross-referencing for maintaining cross-references for section numbers, table numbers, and other document information. This facility provides a `follow CREF (cross-reference) pointer` function to allow the user to view the target of the cross-reference. More interestingly, it contains the `find all fingers` function, which allows the user to see which cross-reference pointers in the document point *to* a particular spot in a document. This forms a rudimentary hypertext capability [NELS67, VAND71a], but requires the high computational overhead of being extrapolated from Scribe, rather than being an editor primitive.

The Document Editor uses the cross reference capabilities to provide functions that manage the task of creating an index for a document. For traveling, the user can follow an index pointer and examine all the fingers pointing to a location, as well as make an index entry, show index symbols, and find all the index symbols containing a particular word.

The Document Editor runs Scribe as an underlying formatting process. The editor itself, EMACS, does not present the formatted text for the user to edit. As discussed in more detail in Section 5, Walker contends that for large documents, one has little interest in anything but the content and the formatting abstractions (as opposed to the actual formatting) during most of the life of the document. However, the Document Editor does provide the functions for compiling those parts of the document that have actually changed, while conforming to the formatting constraints of the entire document (proper page numbers, indentation levels, margins, typefaces). This alleviates the cost of recompiling an entire document because of minor editing changes.

4.6 Syntax-directed Editors

Syntax-directed editors attempt to increase the productivity of the programmer by removing the time-consuming process of eliminating syntax errors. Syntax editors are structure editors that insure that the structure always is constrained to preserve syntactical integrity. Often syntax-directed editors do not merely recognize the syntax and translate the user's actions into linear text, but instead parse the input into an intermediate form that can be used to generate code. Here the editor is both a tool for the programmer and a tool for the compiler/interpreter. We give some prototypical examples below.

Hansen's EMILY

Hansen's EMILY [HANS71] is one of the earliest syntax-directed editors. Rather than typing in arbitrary text, the user creates and modifies text by graphically selecting units of text (templates) that are constructs in a programming language. Text is created with a sequence of selections. The screen is divided into three areas: text, menu, and message. The text area in the upper two-thirds of the screen displays the text under construction as a string that contains the nonterminals (non-atomic entities) of the program, highlighted by underlining. The current non-terminal is enclosed in a rectangle. The menu in the lower third of the screen displays a set of possible replacements for the current nonterminal. The user selects a replacement rule and the system makes the substitution, locates a new current nonterminal, and displays a new set of choices. The message area is used for entering identifiers and also displays status and error messages. Assuming a partial PL/I-type grammar like the following:

```
<STMT> ::= <VAR> = <EXPR>; | IF <EXPR> THEN <STMT> | DO; <STMT*> END;
<STMT*> ::= <STMT> | <STMT> <STMT*>
<EXPR> ::= <EXPR> + <EXPR> | <VAR>
<VAR> ::= id
```

where symbols surrounded by "<" and ">" are nonterminals, an IF statement might be created in the following manner:

The current (boxed) nonterminal is <STMT> and the menu displays the three choices:

```
<VAR> = <EXPR>;
IF <EXPR> THEN <STMT>
DO; <STMT*> END;
```

The user selects the second with a lightpen and gets the expansion:

```
IF  <EXPR> THEN
      <STMT>
```

The current nonterminal is now <EXPR>, and the menu displays the possible expansions for this. Subsequent derivations to arrive at the appropriate IF clause are:

```
IF  <VAR> THEN
      <STMT>
IF FIRSTTIME THEN
      <STMT>
IF FIRSTTIME THEN DO;
      <STMT*>
END;
.
.
.
IF FIRSTTIME THEN DO;
       FIRSTTIME = FALSE;
       SYMBOLS = NULL;
       ENDTIME = DAYMINUTES + 10;
END;
```

Since a syntax imposes a hierarchical structure on text, EMILY can be used for any hierarchical text structure. Each selection from the menu generates a node with space for one pointer for each nonterminal in the replacement string. When a nonterminal is replaced, the corresponding space is filled in with a pointer to the node generated for the replacement. Each nonterminal thus generates a subtree of nodes that is presented on the display, through a tree-walking display routine, as a string of text.

As in NLS, the user can change the view of the text, so that the string generated by any nonterminal is represented by a single identifier called a **holophrast**. For example, the IF statement above could be displayed with all text generated from the <STMT*> represented by a holophrast. In larger programs, this feature means that the user can view the structure of the text without viewing the details. Alternatively, the user can descend into the structure and view the details in full.

Text is also modified in terms of its structure. The text represented by any holophrast can be deleted, moved, or copied. When text is deleted, it is not destroyed immediately, but

is automatically moved to a special system fragment called *DUMP*. If a mistake is discovered before the next text modification is made the deleted text can be retrieved from this dump.

EMILY is a pure syntax-directed editor. Statements are derived by the menu-picking scenario down to the lowest level, for example, the identifier. This makes the editing awkward, since the user must often traverse long derivations to type in a simple identifier or assignment statement.

Cornell Program Synthesizer

Much work in individual areas was done after EMILY, most notably the Mentor [DONZ75, DONZ80] tree-manipulation and programming environment, the CAPS diagnostic programming system [WILC76], and the INTERLISP Programmer's Assistant [TEIW77]. The Cornell Program Synthesizer [TEIT81a, TEIT81b], running on both the Terak (LSI-11 based) personal computer and the VAX family of computers, combines many of the ideas from these and other projects into a syntax-directed editor and programming environment for PL/CS, and more recently, PASCAL.

In the Synthesizer, designed for simple terminals which use cursor keys as the only locator device, the user types textual commands that represent the set of possible expansions of the urrent nonterminal. The set of possible commands can be displayed in an optional window so that the user need not memorize the command sequences. The synthesizer differs markedly from EMILY in that it is not a pure derivational syntax-directed editor. Rather, the synthesizer is a hybrid between the traditional structure editor and the character-string text editor. Thus, common elements such as identifiers, expressions, and assignment statements do not have to be considered as elements of a tree structure, nor do they have to be edited and stored as such.

The user is presented with three types of high-level entities. **Templates** are program constructs which need to be filled in. **Placeholders** are tags in the template describing the parts that need to be completed, and these are the only parts of templates that can be altered. **Phrases** are pieces of text, not structure, that are typed in to replace placeholders.

To start a PL/CS program editing session, the user types **•main** followed by a carriage return to obtain the template for a PL/CS main program.[14] This template is of the form:

```
/* comment */
file-name: PROCEDURE OPTIONS ( MAIN );
  {declaration}
  {statement}
  END file-name;
```

The user can position the cursor at the placeholder **comment** and type a phrase containing the text of a comment. Now the user positions the cursor at the placeholder for the nonterminal **declaration**. Since this is a nonterminal (indicated by the braces), the user must select an applicable template for further derivation. At this point, the user can type **•fx** for a fixed variable, **•fl** for a float variable, **•bt** for a bit variable, **•ch** for a character variable, or **•c** for a comment. For our example we will choose **•fx**. This expands to the template:

```
DECLARE ( list-of-variables ) FIXED [attributes];
```

The cursor is moved to the **list-of-variables** placeholder and a phrase containing the name of the variable is typed in. This name, typed in as text, not as structure, is parsed for syntactic correctness upon pressing carriage return, and is stored and manipulated as text. If an illegal variable name had been typed, this phrase would be highlighted in reverse video and flagged

[14] •, long, clip, delete, left, right, up, down, and diagonal are function keys on the synthesizer keyboard.

internally. If the attributes are not inserted, the square brackets indicate that default values will be used. The declaration nonterminal is now completely defined, and the user moves on to expand the statement nonterminal, for which there are 13 possible templates. Typing **•ie** would generate the template

```
IF ( condition )
  THEN ▣tatement
  ELSE statement
```

(The box here indicates the current cursor position.) Typing **•p** at this position would generate the PUT template, giving

```
IF (condition)
  THEN PUT SKIP LIST ( ▣ist-of-expressions );
  ELSE statement
```

The user could then type a phrase like "**'min = ',beta** " to fill in the placeholder.

The user uses the **left, right, up,** and **down** cursor keys to traverse the structure. In fact, the key names do not represent the true functions attached to those keys. **Right** and **down** both move the cursor forwards through the program; **left** and **up** move it backwards through the program. Rather than moving character by character, these keys move the cursor one program element (template beginning, placeholder, or phrase) at a time. **Left** and **right**, additionally, stop at each individual character in a phrase. In the expanded template like the one above, the cursor would stop at the underscored places when using **up** and **down**:

```
IF ( alpha < beta)
     THEN  PUT SKIP LIST ( 'min = ',beta );
     ELSE  statement
```

and at these underscored places when using **left** and **right**:

```
IF ( alpha < beta)
     THEN  PUT SKIP LIST ( 'min = ',beta );
     ELSE  statement
```

The two-key sequence **long down (up)** moves the cursor to the next (previous) structural element of the same level. Other keys move the cursor to the nearest enclosing structure template and to the beginning of the program.

Insertion and deletion are based on the pick, put, and delete buffer concepts. The user positions the cursor at an appropriate template or phrase, and then issues the **delete** command to delete that template (including, of course, all sub-templates) or phrase and store it in the delete buffer. Similarly, **clip** will store a copy of the selected entity in the clip buffer, but not delete the original. The **insert** command allows the reinsertion of the deleted or clipped text at the current cursor position. In the above example, if the cursor were positioned at the P in "PUT," the sequence **delete, down, insert** would result in the program segment

```
IF ( alpha < beta )
  THEN statement
  ELSE PUT SKIP LIST ( 'min = ', beta );
```

Correcting mistakes can only be done by preserving structural integrity. Assume the following incorrect code segment:

```
/* compute factorials from 1 to 10 non-recursively */
a = 0;
DO WHILE ( a < 10 );
  a = a + 1;
  fact = 1;
  PUT SKIP LIST ( a,' Factorial =' );
  temp = a;
  END;
DO UNTIL ( temp = 1 );
  fact = fact * temp;
  temp = temp - 1;
  END;
PUT SKIP LIST ( fact );
```

The traditional programmer, realizing that the END of the DO-WHILE loop should properly come at the end of all of this code (nesting the DO-UNTIL and the PUT SKIP LIST), would move the END statement to the end of the code with a single **move** command or a **delete/put** sequence to achieve:

```
/* compute factorials from 1 to 10 non-recursively */
a = 0;
DO WHILE ( a < 10 );
  a = a + 1;
  fact = 1;
  PUT SKIP LIST ( a,' Factorial =' );
  temp = a;
  DO UNTIL ( temp = 1 );
    fact = fact * temp;
    temp = temp - 1;
    END;
  PUT SKIP LIST ( fact );
  END;
```

In a syntax-directed editor, since the END is part of the DO-WHILE template, it cannot be separately moved. Instead of moving the END forwards, the equivalent backward move of the two following statements must be done. To perform the desired alteration, the user would have to position the cursor at the start of the DO-UNTIL template, press **long delete**, move the cursor to the last element in the list of structures to be moved (the PUT SKIP LIST (**fact**) statement), signal completion of the selection by typing •, move the cursor to the structural element after which the new part should be inserted (the **temp = a;** phrase), press carriage return to open a **statement** placeholder, and issue •**ins** DELETED to position the desired text in the desired spot. While this is certainly more complicated than the traditional method, the interface is partially to blame. A pointing device that would easily allow selection of elements and extension by structural or contiguous units would eliminate many of the keystrokes above. Even without the pointing device, one could imagine extending the starting or ending portions of a template to encompass contiguous statements.

Even if many syntax-directed editing techniques are nominally longer than traditional techniques, the excess time must be weighed against the time saved by ensuring that a program is syntactically correct every step of the way. One major time-wasting operation that is avoided

is the backmapping of frequently inscrutable syntax-error messages to the source lines, all too often a heuristic and frustrating process. Indeed, an important contribution of the Synthesizer project was the concept of the syntax-directed editor as an integral part of a programming environment. The synthesizer is not typically used to create text files which will later be passed to a standard compiler, but rather as an editor which will create a representation of a program suitable for online interpretation. The Synthesizer allows the user to run a program and watch the cursor step through the lines of code as they are being executed, much like the "bouncing ball" familiar to cartoon watchers. Information hiding (such as seeing only the comments or top-level templates) still allows single-step viewing of the program in which the cursor jumps from one visible high-level unit to the next; the user does not have to watch the low-level details, for example, the inner workings of a loop. Uninitialized variables are flagged, type checking is enforced interactively, and duplicate declarations are prohibited, all at edit time, rather than compile time. Invalid phrases are highlighted as soon as the user types them in. A syntax-directed approach avoids the time-consuming backmapping error messages from a batch compiler to the proper lines in the source file by generating the error messages interactively, with the offending program components highlighted. Programs are incrementally compiled, allowing the user to re-edit and experiment with small parts of a program without waiting for an extensive recompilation. In fact, the approach taken with the Synthesizer allows the suspension of program state, the correction and incremental compilation of a portion of the program, and the resumption of the program.

Templates can be input only in a structurally sound manner, while phrases, typed textually, are allowed to be erroneous. When editing, the user does not need to expand all nonterminals or remove all errors in phrases. An incomplete or erroneous program can be run at any time. However, these irregularities are highlighted from the moment they are input until the moment they are corrected; the synthesizer relaxes some of its constraints, but warns the user accordingly. In both cases, the program will run normally until the error or unfinished program construct is encountered. When this is encountered, the user is free to correct or insert the code and continue the execution.

The program is stored as a combination of a parse tree for the templates, and as actual text for the phrases. The prettyprinted code that the user sees is actually an interactively generated view of the internal data structure.

Currently, a Synthesizer Generator is being developed which will allow a complete syntax-directed editor to be generated from a formal description of the syntax. We point the reader to the GANDALF project at Carnegie-Mellon University [HABE79, NOTK79, FEIL80, MEDI81], for a description of a similar syntax-directed editor and editor-generator project.

Frasers's *sds*

Fraser's *sds* is a general structure editor driven by a grammar that describes a hierarchical data structure. Our interest in it is due to the stress that has been put upon imposing a syntax on targets that are not necessarily programs, and upon the generation of the editor from a procedural description.

The user-viewable part of *sds* is a screen editor which displays a current record of some tree structure. The cursor keys **down**, **up**, **left**, **right** and **home** allow the user to move down to a node field, back up, left or right to adjacent fields, or to the root of the structure. Like the Cornell Program Synthesizer, other commands consist of typing a period followed by the name of a nonterminal. This causes the editor to allow the user to enter the first field of this new nonterminal. The user can either enter another nonterminal designation or, if applicable, simply

type a string that will become a terminal or leaf node. As well, *sds* provides target-independent commands such as `.w (.r)`, which write (read) a subtree to (from) a file, `.hide(show)`, to suppress(exhibit) detail of a subtree, `.pick`, which saves a pointer to the current node, and `.put`, which substitutes the current node with the previously picked node.

The target-specific editor is written using a formal syntax description similar to that used for the YACC compiler-compiler of Johnson [JOHN75]. The entire grammar for an *sds* binary tree editor is captured in one line[15]:

```
tree = value tree tree : dotree(value,tree,tree2)
```

The phrase before the colon is the grammatical description of `tree`, the only production in the grammar of binary trees. The portion after the colon is SNOBOL4 code to perform an action (`tree2` is the second argument named tree, `treen` would be the SNOBOL argument for the *n*th `tree` token in a production list). In this example, the `dotree` subroutine contains SNOBOL code to display the value and the two subtrees in graphical form. Note that to change the representation of a binary tree node to one in which the value lay between the two tree pointers, one would simply have to change the production to transpose the words "value" and "tree":

```
tree = tree value tree : dotree(value,tree,tree2)
```

Similarly, the `dotree` routine could be changed to store the binary tree in a disk-oriented form or to print it in an indented representation; the actions are independent of the creation routines of *sds*.

A document editor has also been written in *sds*. Of course, this implies the construction of a hierarchical grammar for a document, coupled with action rules for each production. A sample grammar for a small document system looks like:

```
paper = title sect : center(title) nl nl generate(sect)
sect = header pp sect : header nl nl put(pp) generate(sect)
pp = text pp : break(text) nl generate(pp)
```

To the right of the colons lie production-specific SNOBOL code. We are concerned here only with the productions to the left of the colon.

To use this editor, the user would enter textual commands to create various levels of the subtree as follows. The prompt line gives the user an idea of location in the structure. The last item on the line is the current field (item on the right side of the production), while the preceding items are the types (items on the left side of the production) which brought the user to that field, i.e., the successive nodes of the tree branch. The root name `paper`, is implied at the beginning of each line.

First, the user types .paper, telling *sds* to begin a node of type `paper`, the root of the structure.

```
prompt:
user: .paper
```

The next prompt asks the user to type in a title and go to the next part of the production:

```
prompt: title
user: Interactive Editing Systems
```

sds is now ready to perform the `sect` production, but requires the user to issue the explicit command to create the section:

[15] Examples are adapted from [FRAS81].

prompt: **sect**
user: **.sect**

Having created the **.sect** record, the system prompts the user to fill in the header field:

prompt: **sect header**
user: **Introduction**

The user is now prompted to create a **.pp** record, and again must issue an explicit command:

prompt: **sect pp**
user: **.pp**

The user is prompted to enter text. In this mode, he is provided with a target-dependent text editor based on the Irons model:

prompt: **sect pp text**
user: **The interactive editor has become an essential...**

Upon terminating that paragraph, the user is prompted to create another, as the **pp** production is recursive:

prompt: **sect pp pp**
user: **.pp**

The user then types in the appropriate text:

prompt: **sect pp text**
user: **Though the editor has always been deemed...**

The command *up* goes up one level in the structure. This cause a production (**pp = text pp**) to be completed and an action to be performed; in this case, formatting of a paragraph:

prompt: **sect pp pp pp**
user: **up**

We go up one more level of the tree, formatting the first paragraph.

prompt: **sect pp pp**
user: **up**

While the user is entering text, *sds* is performing syntax-checking, flagging and prohibiting invalid structure at any point in the document.

Initial reaction to document creation by structure centers on the apparent "wordiness" necessary to get the job done, but Fraser contends that the explicit structure is almost identical to what one does implicitly with a compiler-based document language. In fact, the **.w** command would store the above paper as

```
.paper
Interactive Editing Systems
.sect
Introduction
.pp
The interactive editor has become an essential...
.pp
Though the editor has always been deemed...
```

A third application for *sds* is as a picture editor for simple line drawings. The structure editor, using the small, 6-line grammar described in [FRAS81], would create the multicolor letter "T" with the structure:

```
.branch
.color
blue
.line
0,20 20,20
.color
.red
.line
10,0 10,20
```

Other grammars used by *sds* include one for a subset of C [KERN78c].

4.7 Word Processors

WordStar

WordStar [MICR81] is one of the most popular word processing programs available for home computer systems. It runs on a variety of systems under the CP/M operating system, using the CP/M file system to maintain its files.

The conceptual model of text in WordStar is the quarter-plane of the Irons model. Control key combinations (special prefix characters are used as software shift keys to provide a large set of commands), function keys and cursor keys are used for command specification. WordStar combines the quarter plane model with a "virtual typewriter" model. The user is presented with a ruler line that simulates tab rack and margin ruler on conventional typewriters, and with commands to move virtual margin keys forward and backward on this ruler line. WordStar divides the file into logical pages that default to contain 55 lines (the number of lines on an 8-1/2-x-11 page, excluding margins). Most importantly, WordStar provides modest interactive editor/formatter capabilities for justified, monospaced text. As the user types in text, the lines are automatically justified. When text is changed, rejustification is not automatic, but is done on a per-paragraph or per-document basis by user command.

The screen is set up to provide extensive feedback to the user. The first line is a status line: it presents the filename of the document, the current page number, and the current line and column at which the cursor points; as soon as the cursor is moved, these latter values are changed. As well, the beginning of this line is used to echo the typed command. For instance, as in Figure 23, if the user types CTRL-Q on the keyboard, the textual representation ^Q is shown on the screen. The next few lines on the screen (above the ruler line) represent the current options. Here, since CTRL-Q was typed, the ^Q prefix options are displayed in the help area. The user then chooses one of the ^Q suffixes, which represent commands. A more sophisticated user can avoid this extensive prompting in two ways. First, if the entire command, say CTRL-QF is typed together quickly, it is executed without displaying the ^Q options. More explicitly, the user is given commands to change help levels. These help levels range from displays for the novice, containing complete options to those for the expert, containing no options at all. The full set of WordStar commands is shown in Figure 24. For errors, WordStar makes sure that the user has noticed the error by requiring an acknowledgment – by default the ESC key – to resume operation.

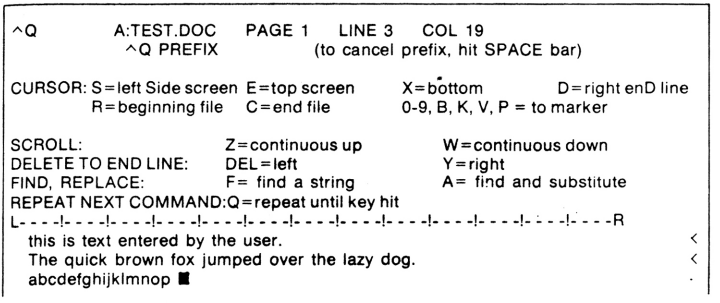


Figure 23. *WordStar screen [MICR81]*

As in the Irons model, editing is done on the displayed viewing buffer/editing buffer by driving the cursor around and typing. WordStar offers both insert mode and typeover mode.

A major flaw of WordStar is the lack of an undo facility: once a command is executed, it cannot be reverted. This reduces the freedom of experimentation that an author should have. The only recourse a user has is "undoing" an entire session with an abort command.

A problem with WordStar, and with most microcomputer editors, is lack of both main memory and disk space. WordStar, for instance, has its own paging routines to bring parts of documents in and out of memory. If the disks are of reasonable capacity, this offers no problem. However, for small systems with floppy disks and consequently small disk capacity, the amount of the disk needed for paging leaves little room for document storage. This causes, in some systems, the unfortunate situation in which a document that is being edited cannot be stored back on disk.

CPT 8000

CPT is a representative example of a commercial standalone word processing system. The Disktype 8000 has a page-size, monospace display, and two floppy disks to store files. CPT was the first word processing system to offer a 8-1/2-x-11 white screen with black characters, simulating a piece of paper in a typewriter [SEYP79]. In fact, the typewriter metaphor is consistently applied. A few lines up from the bottom of the page is the **typing line**, meant to simulate the paper bail on the platen of a typewriter. Input takes place on the typing line only.

No standard cursor keys exist on the CPT. Rather, the space bar moves the cursor forward on the typing line and the backspace key moves it backwards. There is no need for up and down cursor movement, since it is the document that travels up and down past the typing line. Keys are provided, therefore, to scroll the document up and down. Margins are set by moving right and left markers on the typing line. Five other keys specify character, word, line, paragraph

and page elements for commands like **delete**, **skip**, **move**, and **insert**.

CPT provides three input modes. **Manual** mode simulates a typewriter; when the user reaches the right margin, a bell rings. **Wraparound** mode provides automatic carriage returns when the right margin is exceeded. **Hyphenation** mode performs automatic hyphenation when a word reaches a system-defined "hot zone," using an algorithm aided by an exceptions dictionary.

One interesting feedback mechanism of CPT is its error message facility. In the center of

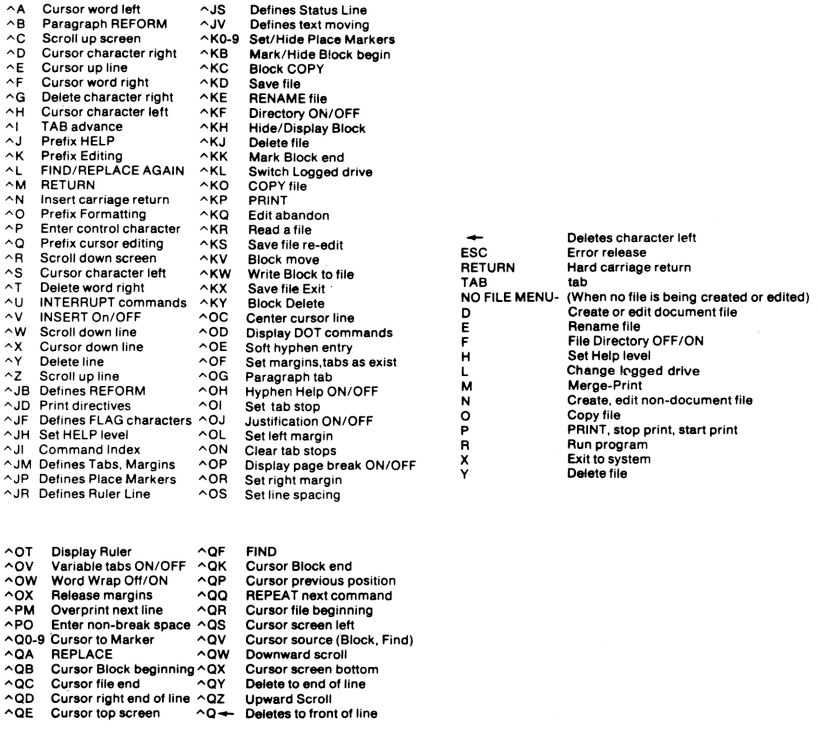


Figure 24. *WordStar commands*

the status line at the bottom of the screen is a 20-character area reserved for error messages. Rather than having terse error messages in this area, and as an alternative to removing some of the possibly offending text from the display to make room for a wordy error message, CPT rolls the lengthy error across this area like a captioned bulletin at the bottom of a television screen.

NBI System 3000

The NBI System 3000 is another popular commercial word processing system. It has a standalone processor, with file storage on floppy disk. Its conceptual model is very similar to that of WordStar described above. The interface uses a combination of option sheets and function keys; the display is a mapping of a screen-sized viewing buffer to a full-screen viewport. The user alters the documents by driving a cursor around the screen with cursor keys, overstriking characters or using appropriate function keys to effect the changes. Like WordStar, the NBI System 3000 supplies the user with option sheets to show available operations. It does not however, offer the help level commands of WordStar. In some cases, the user can operate without calling up an option sheet at all, and in other cases, as in WordStar, if the user is "faster" than the option sheet, it is not called up at all.

NBI presents an interesting alternative to the insert mode vs. overstrike mode "controversy." When the cursor is positioned over a character, typing will overwrite that character. When the cursor is positioned over a space, typing will invoke insert mode and all characters to the right of the cursor will be pushed to the right as necessary. NBI provides an extensive search and replace facility, allowing the user to perform case-insensitive searches and replacements on a case-by-case basis.

Along with these advantages are several inconsistencies. The commands set is not consisent. The `line delete` command will delete the line in which the cursor is positioned, regardless of where the cursor lies in that line, while the `word delete` command will only delete a word if the cursor is positioned at the first character of that word. Although complete region selection and associated copying, moving, deletion, and storage are available, NBI provides no feedback as the areas are selected.

4.8 Integrated Environments

RIG, Apollo

The Rochester Intelligent Gateway (RIG) user interface [LANT79, LANT80] and the similar Apollo Aegis user interface [APOL82] are two examples of a relatively new trend in editing systems, one in which the editor is an integrated part of the interface presented to the user, rather than a user-invoked utility program.

Both the RIG and Apollo systems are based on the concept of a display or window manager as the primary interface to the system. These display managers gives the user the ability to create windows on the display surface, move these windows around, and change their size. On the Apollo these windows can overlap; in RIG the windows do not overlap but simply partition the display screen (see Figure 25).

As in Star, the windows are meant to simulate pieces of paper on a desk. More specifically, the window shows a rectangular portion of a two-dimensional **pad** (see Figure 25), an unbounded quarter-plane of text. Editing functions much like those in Irons-model editors are supplied to manipulate items in the pad.

So far, the systems sound much like a standard multi-window editor. In fact, one of the two types of windows, the **editing window**, fits that model exactly. The other type of window, the **process window**, is an editing window with an arbitrary user or system process attached to it. The process window has both a large **output pad** window on top and a smaller **input pad**

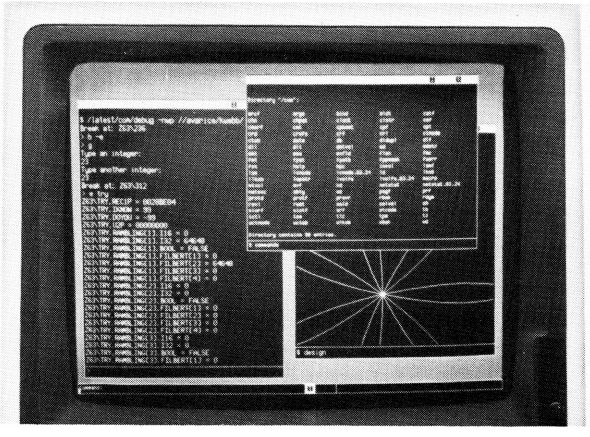


Figure 25. *RIG/Apollo Pads [APOL81]*

window directly underneath. For interactive processes, input is typed into the input pad and, much like a typewriter, when a carriage return is pressed, this text scrolls up over the "typing bar" into the output pad. In addition, output from the interactive process is written directly to the output pad. The result is a complete transcript, simulating output on a hardcopy terminal, of the interactive session. Multiple windows on the screen may be active at once; the user may be reading electronic mail in one process window, editing the file referenced in the mail in an editing window, and compiling that file in another process window. Returning to our editing concerns, we note that the output pad allows all editing functions (though often the system makes the pad's contents read-only so that they can be perused or selected but not altered). The user can now travel through the entire contents of an interactive session, and actually select previous input and output from the output pad and use it as input elsewhere in the system. For instance, the user might select a code fragment directly from the electronic mail window and insert it in the open editing window.

The major use made of this model is to tie to a process window the operating system command interpreter process. Now, the user types operating system commands into the input pad, and both this input and the output of the system programs invoked are stored in the output pad. Commands can be reexecuted simply by selecting their text in the output pad and inserting it in the input pad. Output from a program that is lying in an output pad can be selected and stored in a file. In fact, an entire output pad can be saved as a file if the user wants a transcript of an interactive session.

The ramifications of this concept are wide-ranging, much like the concepts imparted by the Star interface, but in the framework of a general-purpose computing system, rather than an office automation system. The editor does not create a preemptive environment [SWIN74], in which functions normally available to the user are suddenly cut off. Since the editor is now above

the command interpreter (rather than being an applications program invoked by the command interpreter), the user can freely issue system commands interspersed with editing commands. No longer does the user have to leave the editor to do something as simple as reading electronic mail or listing the files in a directory; the user simply switches windows momentarily, executes the appropriate command, and returns to the previous window.

Smalltalk-80

Smalltalk-80, a research product of Xerox PARC's Software Concepts Group (originally the Learning Research Group), provides an even more integrated environment than described above. In fact, the paradigm of overlapping windows was developed as part of the Smalltalk-80 project [LRG76]. Composed of an object-oriented programming language and an integrated user interface, the Smalltalk-80* [GOLD82, GOLD83] system currently runs on several Xerox personal workstations (the Xerox 1100 Scientific Machine and the Dorado) with bitmapped raster displays and three-button mice (see Figure 26). The aim is to give the user an interface in which editing commands are always applicable and other capabilities that the user desires are at hand as well. Anything on the screen may be edited: document text, commands, program code, and so on. The user does not become trapped in modes (as in SOS above), but always has a full range of choices at any point in the editing session.

The conceptual model provided by the Smalltalk-80 user interface [GOLD82] is one of views, represented as labelled, rectangular, possibly overlapping pieces of paper on a desktop. A view is a particular way of displaying the information of a task or group of tasks for user inspection, alteration, storage, and retrieval of information. The views most used are standard system views, in which operations to alter size, location, label and level of detail are defined.

Menus are the other important entity in the Smalltalk-80 user interface. They exist in two varieties: fixed and pop-up. A fixed menu is a **subview** of a displayed view; the user moves a cursor over the menu items with a mouse and selects an item by pressing the leftmost mouse button. This action highlights the selection. Releasing the button invokes the selected command. A pop-up menu appears directly under the cursor when the user holds down one of the other mouse buttons. As the cursor is moved around the menu, the item underneath the cursor is suitably highlighted. Upon releasing the button, the item is selected and the menu disappears.

Since the screen space available to present a view may not be large enough to contain all the information that needs to be presented, Smalltalk-80 provides the **scroll bar**, a special type of menu that allows the user to select what portion of the view is to be made visible. It supports three general operations: `scroll up`, `scroll down`, and `jump`. The scroll bar is displayed as a vertical white rectangle that appears out to the left of a view's rectangle when that view is being used. This white rectangle represents the document continuum. Inside this rectangle is a smaller, gray rectangle. This represents the viewing buffer of the document, the amount of the document that is currently being viewed. For example, in Figure 27a, the gray bar indicates that about the last half is being viewed; in Figure 27b, the first half is being viewed; and in Figure 27c, the entire document is being viewed. To jump to a particular place in the document, the user puts the cursor in the gray rectangle, holds down the leftmost button, and drags the gray rectangle up and down by moving the mouse. Moving the rectangle simulates changing the placement of the viewing buffer on the document. Upon releasing the button, the view is changed to reflect this viewing buffer placement. To scroll down (up) the user places the cursor on the left (right) of the gray rectangle, and the cursor automatically changes to a down (up) arrow. Upon pressing the leftmost button for downward scrolling, the line of text closest to the

*Smalltalk-80 is a trademark of the Xerox Corporation

Figure 26. *Smalltalk-80 Screen and Mouse*

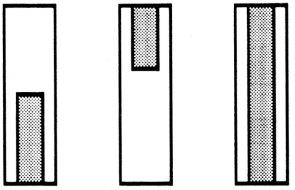


Figure 27. *Smalltalk-80 Scroll Bar*

line of text at the top of the view is moved to the cursor position. Upon pressing the leftmost button for upward scrolling, the line of text closest to the cursor is moved to the top of the view. Other menus include **confirmers**, which allow a user to select one of two displayed choices, and **prompters**, which allow the user to "fill-in-the-blank" in response to a question or message.

To edit text in a system-specified view, the user makes extensive use of the mouse and the supplied menus. Pressing and releasing ("clicking") the leftmost button on the mouse causes a caret to appear in the inter-character gap closest to the cursor. This essentially selects a zero-length string. If the user holds down the leftmost button, all the characters between the initial caret and the current position of the cursor are highlighted in reverse video. Releasing this button causes the highlighted text to become the active selection. Two clicks on the leftmost button while the cursor remains stationary select the word on which the cursor lies, unless the cursor is at the beginning or end of the document, in which case the entire document is selected, or unless the cursor lies just after (before) a left (right) parenthesis, square bracket, angle bracket, single quote, or double quote, in which case the text between the delimiter pair is selected.

Replacement fits naturally with the concept of selection. Inserting characters is simply a form of replacement. The user moves the cursor to the gap in which the string is to be inserted, does a single click to get the caret, and begins typing, which causes characters to be inserted at this spot: the user is essentially replacing the zero-length string with the string being typed (see Figure 28). If a larger area is selected, the typed-in text replaces the selection (the first typed character causes the deletion of the selected area.)

Copying and moving are done analogously. The proper selection is made and either the **copy** or the **cut** button is selected from the pop-up view menu controlled with the middle mouse button. Both commands store the selected text in a paste buffer; **copy** leaves the selected text

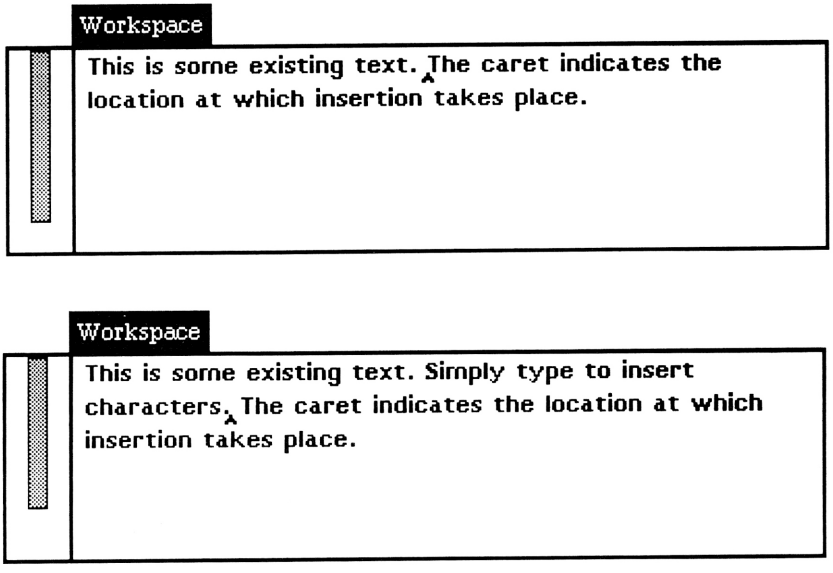


Figure 28. *Insertion in Smalltalk-80*

unscathed while **cut** deletes it. Now the user is free to perform any action; the **copy** and **cut** are finished. When desired, the user simply selects a destination point, selects the **paste** command in the menu, and the proper insertion is made. If the user has just selected and replaced text, **again**, another command in the pop-up menu, will hunt for the next occurrence of the pattern that was selected and replace it with the text used for the replacement. If the user selects text and then issues the **again** command, the system will simply search forward for the selected pattern, and if found, select it so the user may cut it or replace it (see Figure 29). An **undo** command allows the reversal of the previously issued command.

Most Smalltalk-80 commands are issued by either clicking a mouse button, or by the dual motion of first pressing a mouse button to select a menu item and then releasing the button to invoke the associated command. When a command is not available this way, the user simply finds some empty view space, types in the appropriate command, selects it, and then issues the `doIt` command. Later on, one may come back, edit that command in the normal way, and reissue it. Thus commands are edited as easily as text; indeed, commands are simply text. The `printIt` command is identical to `doIt`, with the exception that any output result is placed immediately after the command text to which `printIt` was applied and this result is automatically selected.

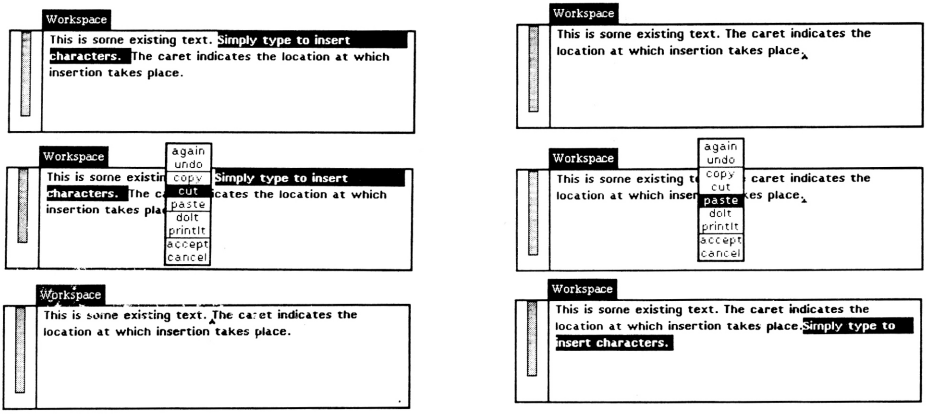


Figure 29. *Cut and Paste in Smalltalk-80*

In a, the user selects the text to be moved. In *b*, the user holds down the middle mouse button and the pop-up menu appears. The user moves the cursor over the **cut** button, which is appropriately highlighted. In *c*, releasing the mouse button causes the selected text to be removed. In *d*, the user selects the re-insertion point with the left mouse button. In *e*, the user holds down the middle mouse button and the pop-up menu appears. The user moves the cursor over the **e paste** button, which is appropriately highlighted. In *f*, releasing the middle mouse button causes the previously cut text to be pasted in at the selection point.

The environment supports a wide range of tools for the development of Smalltalk-80 programs. While most are beyond the scope of this paper, we mention **browsers** as an interesting method for traversing hierarchies. Objects in the Smalltalk-80 language are built hierarchically. For instance, a **number** is a *class name* that is in the browser *class category* **numeric objects**. Similarly, the operation of addition is a *message selector* that is in the browser **numeric-number** *message category* **arithmetic**. To find information about these quickly, the user simply selects

numeric objects in the class category subview in the browser, which brings up the appropriate classes in the class name subview. The user then selects **Number** in the class name view, which brings up the appropriate method categories in the message category subview. The user then selects **arithmetic**, which brings up all appropriate messages in the message selector subview. Choosing one of these message selectors, for example **abs**, causes the Smalltalk-80 procedure for that method to be brought up in the editing view below (see Figure 30a). The user is free to edit this procedure using the editing facilities described earlier. Pointing to an item in a subview further up the hierarchy replaces the subviews below, and allows the user to browse through new definitions. For instance, as in Figure 30b, selecting **truncation and round off** in the class category subview causes a new message category subview to be generated and the message selector subview to be changed. Subsequently, selecting **truncated** causes a new procedure to appear in the editing subview.

The browser is important not only for its convenient techniques for tree traversal, but for its notion of letting a user browse through an entire collection of information, examining and editing it at will. A browser-like interface would be attractive in other environments as well, such as that of examining a file system hierarchy or traversing a hierarchically-structured menu system. Indeed, the access to local and remote files is provided in the Smalltalk-80 system through appropriate browsers in which filename patterns are specified and filenames are selected for reading, retrieving, or editing.

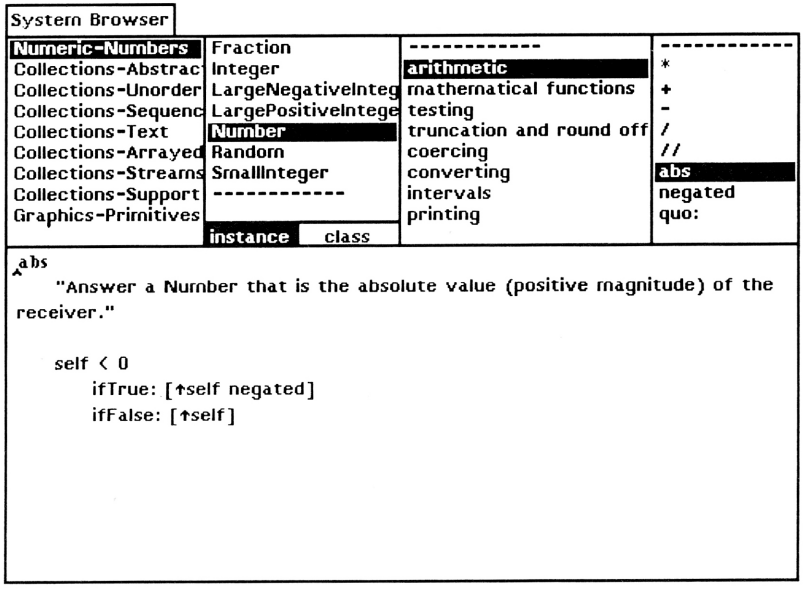


Figure 30. *Browser*

5. ISSUES

5.1 The State of Editor Design

While much is written about "the desirable human interface," most of it (unsupported) personal opinion, very little experimentation in determining the optimal editor interface has been done. Typically editor design is based not on concrete experimental results, but on market pressure to design systems that conform to today's worn technology (such as 24-x-80-character terminals, and half-duplex communication to timesharing systems). Rather than concentrating on desired functionality and ease of use, the editor designer is forced to devote large amounts of time to molding the user interface to the constraints of particular classes of limited input and output devices, producing a far-from-optimal interface.

In our editing model, the lexical phase of the command language processor, which composes tokens from lexemes, is followed by a syntactic phase, which parses sentences of these atomic tokens. In principle, we want each token's appearance and meaning to be unambiguous in all contexts, and its user image to be unique, easily remembered, and unobtrusive. For typed command languages, this is not the case: the user must correctly spell or abbreviate the needed tokens, usually from memory, and the system must be in the appropriate "command" mode so that command tokens are accepted as such, and not as literal text. In control-key interfaces, tokens are composed by overloading the alphanumeric keyboard with control or prefix keys to form cryptic combinations.

If tokens are atomic entities unto themselves, why do we need a lexical component to compose them at all? In fact, the token composition phase is one of the most treacherous parts of a user interface, and is, for all practical purposes, unnecessary with modern technologies. The pure function key interface, for example, assigns a token to a particular key; composition of tokens is not necessary. The flaw in this technique is that the number of function keys grows linearly with the number of tokens; a many-function editor would need a massive, untenable keyboard. One type of interface that begins to satisfy the criteria of atomic tokens without incurring the expense of linear growth of input devices is the menu interface, more explicitly, one that uses pop-up menus in temporary viewports and selection devices like the mouse (see Smalltalk-80 in Section 4). In general, menus supply composed tokens in a form the user easily recognizes. No memorization of tokens is needed, since the necessary images appear as needed; the user simply points to the appropriate image and selects it. The menu presents to the user only those tokens that are viable at any particular time, unlike the function keyboard, which is obliged to have all tokens at all times to avoid overloading. The menus are unobtrusive; when not in use they disappear and only pop up when called. Another interface satisfying this criteria is the Star interface. Star eliminates the problem of overloading commands by having a small set of general-purpose function keys such as **open**, **move** and **find** that provide the majority of tokens needed, a rich set of icons that serve as tokens as well, property sheets and option sheets that provide consistent access to tokens as well as a consistent method for composing new tokens in these sheets, and finally, window-specific menus containing selectable command tokens.

Design techniques pose another problem. Many editors in production today have been designed *by* programmers *for* programmers, and have been foisted upon the general public with little apparent regard for its needs. Many others appear to have been designed by non-programmers for non-programmers, and show little evidence of proper software engineering and language design principles, such as consistent user instruction sets and consistent syntax. Clearly few editors have been designed by rigorous examination of reasonable choices in interface and

functionality, and even fewer are backed by a well-explained conceptual model. Rather, editing design has been ad hoc, with the editor often becoming a potpourri of contradictory techniques and functions, copying and inheriting poor design from previous systems ("we can always change it or write another one"). It is high time that editor designers, like programming language designers, commit their conceptual models and user interfaces to paper before implementation. This requires extensive search of the literature, analysis of alternatives, and experimental validation of ideas, all traditional actions in science and engineering but disappointingly rare in this field.

5.2 The Modeless Environment

There has been much recent interest in so-called "modeless environments" [TESL81]. In fact, the term "modeless" is a bit of misnomer since no system can be truly without modes. What is intended is that modes be minimized, that designers move away from implementing special-purpose context-sensitive states and commands of the type that SOS has. The primary problem with modes is that they lock the user into a specialized and typically highly restricted functionality while in the mode, preempting the use of the normal set of functions and thereby severely limiting flexibility. With current techniques for command specification there is often a second problem, that of assigning different meanings to user actions as a function of the mode, as is the case in overloaded keyboards.

The goal of the modeless editor is to allow the user to have the flexibility to travel and to select operands without having to commit to a particular function and the particular options that it allows. In particular, the postfix form of command specification in which the operands precede the command is more conducive to minimizing modes than the prefix form. The latter is typically used to put the user in a temporary mode and then prompt for the operand(s): for example, **move** puts the user in a mode that requires the user to specify the source and then the destination. This style of guided dialogue, while useful for novices, is often frustrating and annoyingly time-consuming for experts. Furthermore, it enforces sequential specification of multiple operands, without providing the ability to edit them. Worst, of course, is that the user is locked into the dialogue, and cannot leave to browse, or to collect information with which to complete the command (see Tesler's painfully amusing example in TESL81).

With postfix syntax, the user spends most time browsing and selecting without committing to a particular function. When the function is specified, it is executed indivisibly and the user is back immediately in the familiar and universal operand selection "mode," free to browse, to create other window/viewport combinations and to select and revise selection of operands before specifying another operation. It would be possible to allow the user to escape from a temporary operation mode in prefix interaction, but a great deal more status-saving would be required.

In the case in which certain commands require more than a single operand followed by an operator, there are several alternatives in a postfix system: 1) split a compound operation into smaller primitive operations that fit the postfix constraint (as in two single-operand **cut** and **paste** commands to replace **move**); 2) allow multiple selection of operands, although this becomes difficult if order of specification is important; 3) allow the use of familiar editing functions for specifying parameters and for setting attributes by letting the user fill out a form and then execute a command based on this form; and 4) temporarily switch to infix specification. For an example of the last alternative, a **move** command that takes both a source and a destination would be specified in the normal way by selecting first the source and then **move**. Then, the system would enter temporary **move** mode, ask the user to select the destination, execute the **move** command and return to familiar operand selection mode.

The Star system uses the third alternative above, providing property sheet forms that do not preempt the user (as shown in Section 4). To issue the `find` command, for example, the user presses the `find` key, gets the property sheet, fills in the appropriate parameters, and then issues a command that actually does the search. Using the form metaphor, the user has the ability to select information from other parts of a screen as input to the form, and, of course, has the ability to edit the form as well. In fact, the form can be used to simulate a dialogue. As the user fills in particular information or toggles particular attributes, the system can provide further fields to be filled in. If the user goes back and edits one of the fields, all of the field values that depended upon one particular field value might be undone. (See the example of Star `query-replace` in Section 4.) Not only does the form metaphor simulate interactive dialogues, but it obviates the sequentiality and non-editability problems of conventional dialogues. Why then is filling in a form not considered "dangerous?" In fact, form filling is a mode of sorts, yet familiar functions can be used to edit it. Most importantly, the user can leave the context of the form, issue other commands, and return without loss of context.

5.3 Instant Editor/Formatters vs. Batch Formatters

The classical separation between form and content enforced by batch formatting is becoming increasingly less desirable. Space and layout constraints often force alteration of content to make text fit. Furthermore, text can be interpreted in a surprisingly different way when typeset than when it is printed as draft copy on a lineprinter. This, of course, is the reason it is typeset at all. That computer scientists, reporters, copy editors, and even professional printers have tolerated the system of marked-up alterations and specifications on typewritten copy or bad facsimiles of a final typeset galley is a result of economics and not an implicit confirmation of that system. The typesetting conventions that make it easier to understand text in printed form make it correspondingly easier to understand the online form. For all these purposes and especially for complex formatting tasks (tables, equations), interactive formatting is clearly highly desirable.

Yet, there is a strong camp advocating continued use of batch formatting systems, with possible softcopy review, to allow maximum flexibility and power, especially in terms of multiple interpretations of markup tags (which indicate, for example, different document styles and output devices). Allen et al. [ALLE81] advocate the use of softcopy output that is later more precisely formatted by a document compiler. They contend that the interactive user does not need a finely formatted document, but simply one that approximates the final printed result. The interactive system does not need to perform expert formatting; this is left to a batch document compiler. The underlying notion is that no matter how accurate interactive formatting systems can become, those (batch) methods that spend more time will produce higher-quality output.

Still, interactive editor/formatters seem to have compelling advantages over editors that have a separate, editable representation for formatting effects, and certainly over the separate interactive editing/batch compiling method. The ability to experiment with different formats is clearly invaluable to both author and transcriber, providing that there are no serious restrictions resulting from this facility. Having to "program" formatting effects is a mental burden and requires sophisticated, complicated code all too often; debugging a sequence of formatting codes is difficult unless a formatted copy of the same document exists for comparison.

On the other hand, the problem with some interactive editor/formatters, often called "what-you-see-is-what-you-get" editors is that, as Brian Kernighan has remarked, "what you see is all you've got." That is, it is just as uninformative and unhelpful to give a user a view of a beautifully formatted document with no clues as to how or why the formatting was effected as it is to give the user a file laden with complicated formatting codes without the rules for

what these formatting codes will do. However, the stripping of formatting information is not necessary to interactively produce a finely formatted document. Referring back to our editor model in Part I, Section 1, however, we can interpret the finished document page as simply one of many useful **views**, but not the only one to which the user should be restricted. In fact, the property sheet of Star and the margin tags of ETUDE are simply specially tailored views of the document data structure. In particular, structure editing can be nicely done on representations that stress the structure and suppress formatting information – one can rearrange sections in an outline much more easily if only the section headings and the first line of each section are displayed, as in NLS. Also, as Jan Walker has pointed out [WALK81a], it is often important to know why a particular formatting effect is apparent; it is useful to be able to interrogate and alter the higher-level document object specification that caused the effect.

The principle of multiple views, one that has been sorely under-utilized in the hundreds of editors that have been created, shows that a completely reasonable solution to satisfy both camps is to provide whatever views each desires. The batch community might get a view that allows them to edit textual descriptions of formatting, equations and tables, while the interactive community might be given a view that allows interactive specification of tables, equations, as well as of the traditional simple (and local) formatting effects. Except for the additional implementation time, there is no reason to restrict the user to editing of a single interpretation/view; the multiple viewing principle needs to be adopted in systems of the future.

5.4 Structure/Syntax-Directed Editors vs. "Normal" Editors

With the increase in the number of structure editors, several designers have explained the rationale behind what seems at first to be a restrictive concept.

Advocates of structure editors claim that the specification of target data as well-connected, well-defined units enhance's the user's powers of creativity and composition. Engelbart, describing a key idea in NLS [ENGE68], writes:

> With the view that the symbols one works with are supposed to represent a mapping of one's associated concepts, and further that one's concepts exist in a "network" of relationships as opposed to the essentially linear form of actual printed records, it was decided that the concept-manipulation aids derivable from real-time computer support could be appreciably enhanced by structuring conventions that would make explicit (for both the user and the computer) the various types of network relationships among concepts . . . . We have found that in both offline and online computer aids, the conception, stipulation, and execution of significant manipulations are made much easier by the structuring conventions . . . . We have found it to be fairly universal that after an initial period of negative reaction in reading explicitly structured material, one comes to prefer it to material printed in the normal form.

Burkhart and Nievergelt [BURK80] concur with the view that while the structuring seems to be a restriction on the user (especially the novice), who may not want to be forced to keep track of the data hierarchically, the structuring would ultimately be performed anyway, "into chapters and paragraphs, procedures and modules, subpictures and patterns, as the semantics of the data may suggest."

Using his Cornell Program Synthesizer as an example, Teitelbaum claims the value of the syntax-directed editor is that a program being developed is always structurally sound, even if not complete. The use of structural templates eliminates mundane program development tasks. Indentation and prettyprinting are automatic, typographical errors are possible only in user-typed phrases, not in system-supplied templates, and such errors can be easily caught at

runtime. The templates save keystrokes, as one typed command may generate a long template. Placeholders in the templates act as prompts, guiding the user along the proper path. The user never needs to get mired in low-level syntactic detail – the constructs are always conceptualized as abstract units, not as streams of tokens.

On the other side, Woods [WOOD81] claims that a good "standard" editor can do 95 percent of the program editing that a syntax-directed editor can, at much smaller development and computation costs. He claims that syntax-directed editing constrains the user interface, complicating operations that are normally easy in a standard editor. This is true of some operations in the available interfaces today, but is not an intrinsic restriction on future interfaces. He further claims that the syntax-directed approach promotes a multitude of editors. This is only partially true; editor generators such as the Cornell Synthesizer Generator, the GANDALF/ALOE project [NOTK79, FEIL80, MEDI81], and *sds* show that editors for very different targets can have the same basic editing operations. In fact, the regularity of the structure editor introduces the ability to produce formal descriptions to generate special-purpose editors. This parallels the trend to create parser-generators and compiler-generators from formal descriptions. Of course syntax-directed editors certainly have target-specific operations as well; but these operations then need not be supported by separate lexical and syntactical analyzers. Woods's claim that the representation and editing of a program as a parse-tree makes an editor harder to implement is certainly true, yet, again, the syntax-directed editor eliminates the need for a parser all together.

Notkin points out that syntax-directed editing may change the way programming is taught and described [NOTK79]. For instance, "nitty-gritty" details such as placement of statement delimiters like semicolons could be eliminated entirely, since the templates will carry whatever information is necessary to create a structurally correct program.

Morris and Schwartz [MORR81], among others, contend that syntax-directed editors are "profligate consumers of computer resources . . . . Parsing consumes processing power, the parse tree devours storage, and there is no solution but to supply plenty of each." With high-performance personal workstations possessing virtual memory, however, this is no longer a very important consideration.

We believe that given adequate machine resources and a well-engineered human interface (perhaps a two-dimensional syntax with interesting icons), there are many reasons to prefer a syntax-directed approach to editing. However, we know of no formal determination of tradeoffs between "normal" editors and structure/syntax-directed editors. We hope that controlled experiments in this area will result in the necessary data for an objective, informative evaluation of the utility of structure/syntax-directed editors.

6. CONCLUSION – PART II

6.1 Desiderata for Today's Editor

As in programming languages and most computer systems, the "desirability" of the syntax and semantics associated with an interactive editor is largely a matter of individual taste. Often, however, constraints imposed by old techniques and methodologies and acceptance of outmoded technologies (300-baud "glass teletypes," for example) force inferior modes of communication. Without being unduly constrained by the limitations of old-fashioned technology, we suggest our design criteria for an ideal interactive editor within the limits of today's modern technology. Thus, an ideal editor should have:

- A well-defined, consistent conceptual model, rather than a seemingly haphazard organization. The user must be familiar and comfortable with the "philosophy" behind the system.
- Documentation, both online in a help facility and offline in manuals, which explains the conceptual model as well as the details of the user interface and the functions of the system.
- A clear and concise user interface that is easy to learn and to use and that provides consistency across different targets such as text, pictures, and voice. Indeed, a good test of an efficient and pleasing interface is that authors will use the system to compose and revise manuscripts themselves. We believe that the author should not need to involve others (experts or "wizards" to advise, secretaries to make changes) in any phases of document creation or editing.
- An "infinite" undo/redo capability, enabling an author to experiment without concern of loss or damage to a document.
- Fast response time. No noticeable delay should exist for all but the most complex commands.
- Powerful facilities, with few restrictions and exceptions, to make possible everything that one can do to hardcopy with red pencil, ruler, scissors, and tape.
- Facilities that take advantage of computer capabilities to compensate for human limitations. Examples of existing facilities include global substitution of one pattern for another throughout a document, replication of a standard phrase or paragraph, and automatic renumbering of sections or references after (or while) a file is edited. Some of these facilities may duplicate human processes, others may be functions that are available only with the power of a computer.
- User access to shared information and files under controlled conditions (useful for a pool of researchers or documentors working in the same area, or for common access to dynamically updated management information).
- Ability to mix targets, such as text, graphics, programs, and forms, with ease.
- The ability to have multiple contexts on the same display surface, allowing the user to browse through and use a large assortment of familiar utilities and documents in an editing session. The editor should not force the user into a smaller, less powerful environment in which normally provided system functions are preempted. In fact, the editor should be part of a larger, integrated environment, allowing the user, in the middle of an editing session, to obtain information by looking through the file system, to use a desk calculator utility, or to retrieve an electronic mail message or a piece of data from a database system, with transparent return to the editing context.
- The ability to edit a close facsimile of the final composition, layout, and typography of the document without significant impact on computer response time.

6.2 Standardization

Several ANSI and ISO committees (ANSI X3J6 and X3V1, ISO/TC 97/SC 5/EGCLPT) are considering standardization of generic markup languages, text processing languages, text formatting, text editing, and text structures. We consider the agreement upon a standard editor

as unrealistic at this time. What is needed first is a standard reference model for editing (for example, one based on the framework in Part I, Section 1). The acceptance of at least an interim reference model on which to base further research and development of editors will be helpful for a productive interchange of ideas in the editing field.

Another step towards standardization would be the definition of a set of operations, called a virtual editing protocol (VEP) [MAXE81, WILD82], that acts upon any medium, such as text, graphics, or voice. The VEP would not define *how* operations are performed on the medium, but simply what generic operations can be done on all media. The virtual editor for each medium accepts the medium-independent VEP and translates it to medium-dependent operations.

6.3 The Online Community

Just over ten years ago, in our first survey of the field [VAND71b], we concluded:

> Online composition and editing of programs, coupled with interactive debugging, has become an established, cost-effective use of computers. Similarly, minor text editing, such as the correction of typographical errors in memoranda, is cost-effective, since only teletypewriter consoles and minimal service from the CPU are required. In contrast, the imaginative use of computers for online composition and extensive manipulation of free-form text is still in the early stages of experimentation and user conversion. This is due partially to the high cost of CRT terminals that provide the human factors essential to general-purpose editing, partially to the high cost of system resources and implementation time for the sophisticated programs required, and partially to the long time required to wean users from traditional offline hardcopy processes. Hardware prices are coming down steadily, however, and as more users switch to online thinking, creating, and manipulating, the use of computers will become increasingly accepted (and judged cost-effective), hopefully to the same extent that numerical and data processing applications are already considered to be a legitimate use of the computer.

In the ensuing decade, this hope for the large-scale introduction of text processing was realized, as a result of rapidly decreasing hardware prices, the acceptance of word processing systems in business, the introduction of personal computers in hundreds of thousands of homes and offices, and the rapid growth of software for interactive editing. Regrettably, much text processing today is still done by transcription from hardcopy rather than by authors composing and editing online. This is due in part to generally poor interfaces (limited windows on alphanumeric screens and transcription-oriented software), and in part due to cultural resistance: typing, unfortunately, is not a universal skill and is still too often considered an inappropriate activity for executives and other high-level decision makers.

For the remainder of the decade and indeed the century we see an ever-increasing infiltration of editors. In one form or another they will become a fundamental tool of modern communication in all walks of life. They will be the key user interface to the workstations that will be used in the office, the classroom, the home, and any other place in which information is entered, edited and communicated. They will increasingly be used for do-it-yourself, high-quality document production with sophisticated typesetting effects, as well as for online browsing, studying, composing, and communicating.

As terminals and workstations become widely available and personal (i.e., become part of one's office or study furniture that, like a clock radio, are rarely, if ever, turned off) increasingly more of one's daily activities will be done via the computer, increasingly less via traditional paper and mechanical communication. There is, in fact, a kind of "critical mass" phenomenon, in which a knowledge worker switches from hardcopy to softcopy for most purposes, given fulltime

availability of powerful local services and a high-performance information- and resource-sharing network. At that point, the Bush-Engelbart-Nelson visions of online communities will become commonplace, and editors will be the key interface to all manner of (softcopy) manuscript preparation and communication. We expect these editors to be suitable both for stream and structure editing, and for targets as diverse as text, pictures, and voice. The more intelligence the editor has of both the form and content of the manuscripts, the more powerful its capabilities will be.

Just as advances in technology in the past decade have provoked a marked change in editors, so will advances in intercomputer communication, speech synthesis and understanding, and character and handwriting recognition again change the way in which future editors are implemented. Imagine the following scenario:

Families, businesses, and individuals will receive a symbolic computer address much as one receives a telephone number. A user anywhere in the world with access to this address will be able to access the files at that network address as if they were on that user's own machine (subject to any confidentiality and security restrictions imposed). Interdocument links will easily be made by including this user address as the first search criterion in the link address. Multi-person collaborations will become economically and technically feasible, and will make distributed knowledge work an attractive alternative to physical travel. Tymshare's AUGMENT and Nelson's Xanadu* system are ongoing projects to bring these concepts and many other adventurous document organization ideas to the general public. NELS81 provides both an interesting history of the development of hypertext systems and a description of the Xanadu technical and organizational plans.

Each user will have a personal workstation with a high-resolution (several-hundred-points-per-linear-inch) bitmap display packaged in a flat, notebook-style package easily moved about a desk or carried in a briefcase [LRG76, KAY77, GRID82]. Interaction may be done in many ways. A wireless mouse or a touch-sensitive screen will allow for cursor movement. The cursor may be used not only for selection of entities from a menu, but also for drawing proofreader's symbols on a document or for entering text into the document. A symbol recognition program will understand the drawn symbols and perform the appropriate operation. If a symbol is inscrutable or ambiguous, the editor may notify the user using voice output. The user will have the choice of redrawing the symbol, or alternately, vocally inputting the commands (as well as, later, even natural-language text). While currently experimental and unportable, the use of eye-tracking schemes [BOLT81] may allow the editor to determine at what (large) area the user is looking, enabling it to correctly understand commands such as "delete *this* paragraph."

Before the turn of the century, the editing systems are likely to have taken the place of pen, paper, and typewriter – and not only for manuscript composition. For example, banks will have editors with preprinted "forms" that the user fills out using a keyboard or even natural handwriting. Documents will be interactive, compiled on demand especially for the requester. They will be further personalized with online annotation. Most importantly, schoolchildren will learn to both read and write with the editor and a nationwide library of online books.

In fact, much of the technology for the near-term editor of the 1980s is in place. Among the hardware are bitmapped personal workstations, pointing devices, precision laser printers, and digital phototypesetters; among the software are multi-window text and graphics editors, interactive formatters/typesetters, iconographic communications, and modeless environments. The key issue of the 1980s is the willingness of users and manufacturers to discard existing

* Xanadu is a registered trademark.

techniques for even better-researched, better-understood, and better-developed metaphors of user interaction.

In the broadest sense, most actions that people perform are editing operations of one form or another. In moving a car from here to there, making a shopping list, or playing chess, a person modifies or *edits* the state of some entity. In computing, most of the actions that people perform are editing operations as well. It is inevitable that the interactive editor will soon enter a new generation, a generation in which it forms the primary interface to the computer.

POSTSCRIPT

The majority of this document was edited using the *bb* editor running on a VAX 11/780 under Berkeley UNIX 4.1. Some parts of the document were occasionally edited using the Apollo editor, BBN's PEN editor, and Brown's CMS Editor. Besides these, the authors at one time or another have used ed, ex, vi, EMACS, SOS, the Cornell Program Synthesizer, FRESS, NLS, TECO, XEDIT, WordStar, Bravo, and Star.

Formatting was initially done using the TROFF package under UNIX. A revised version was translated to TEX by the use of keystroke macros in *bb* and through hand-translation for some parts. No interactive formatting was available; the authors had to rely on hardcopy printouts from a Varian electrostatic printer-plotter (approximately 5 pages/minute) to see the formatting that the TEX codes had produced. The text consists of approximately 100, single-spaced, 12-inch high pages. With around 50 drafts of the paper over more than two years, we have regrettably used just under one mile of paper to produce a final draft, excluding the reams of paper used in duplicating review copies.

Communication of machine-readable files and electronic mail was done through the *uucp* inter-UNIX telephone network and through the ARPANET.

ACKNOWLEDGMENTS

The authors would like to thank the numerous people who have contributed their time and thoughts to the production of this paper, especially Steve Derose, Doug Engelbart, Joan Haber, Phil Hutto, Barb Meier, Steve Reiss, Katy Roth, Dave Smith, Tim Teitelbaum, Jan Walker, and Nicole Yankelovich. We also thank the numerous researchers whose systems we have used as examples, and apologize for any misrepresentations that may appear. A special mention goes to Trina Avery, for her outstanding copy-editing and proofreading of multiple versions of the document, and to Steve Feiner, for his customary incisive comments and technical questions that resulted in more accurate explanations and descriptions. Rachel Rutherford's meticulous reading pointed out hundreds of places where we could clarify points for the reader; her delightful comments on the final draft were an important barometer of how well we were achieving our goals. Janet Incerpi answered our barrage of questions about and suffered our tirades over TEX. Mike Braca cheerfully produced photographs with virtually no notice. We gratefully acknowledge the reviewers, whose detailed critiques guided us towards a completely reorganized, clearer document. Finally, many thanks go to Adele Goldberg who provided us with patience and understanding beyond the call of duty, immediate turnaround despite our somewhat more sluggish speed, and most importantly, numerous critical and editorial comments that improved the paper.

CREDITS

Bravo, Smalltalk-80, and Star figures courtesy the Xerox Corporation. SOS figures © 1978, Digital Equipment Corporation. XEDIT figures © 1980, International Business Machines. WordStar figures © 1981, MicroPro, Inc. Etude figures courtesy MIT Laboratory for Computer Science.

7. REFERENCES

[Achu81] Achugbue, J.O.,"On the Line Breaking Problem in Text Formatting," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 117–122.

[Albe79] Alberga, C.N., Brown, A.L., Leeman, G.B. Jr., Mikelsons, M., and Wegman, M.N.,"A Program Development Tool," Technical Report RC 7859, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 1979.

[Alle81] Allen, T., Nix, R., and Perlis, A.,"PEN: A Hierarchical Document Editor," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 74–81.

[Apol82] Apollo Computer Inc., *Apollo System User's Guide, Release 4.0*, Chelmsford, MA, April 1982.

[Arch81] Archer, J.,"The Design and Implementation of a Cooperative Program Development Environment," Ph.D. Thesis, Cornell University, Ithatca, NY, August 1981.

[Arno80] Arnold, C.R.C.,"Screen Updating and Cursor Movement Optimization: A Library Package," Technical Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, September 1980.

[Bake80] Baker, R.,"Terminal Crunch," *The New York Times Magazine*, November 10, 1980.

[Bara81] Barach, D.R., Taenzer, D. H., and Wells, R.E.,"Design of the PEN Video Editor Display Module," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 130–136.

[Baud78] Baudelaire, P.C.,"Draw," in *Alto User's Handbook*, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, November 1978, pp. 97–128.

[Beac81] Beach, R.J., Beatty, J.C., Booth, K.S., and White, A.R.,"Documentation Graphics at the University of Waterloo," in *International Conference on Research Trends in Document Preparation Systems*, Lausanne, Switzerland, February 27-28, 1981, pp. 123–125.

[Bilo77] Bilofsky, W.,"The CRT Text Editor NED - Introduction and Reference Manual," R-2176-ARPA, Rand Corp., Santa Monica, CA, December 1977.

[BBN73] Bolt Beranek and Newman Inc., *TENEX Text Editor and Corrector Manual*, Cambridge, MA, October 1973.

[Bolt81] Bolt, R.A., "Gaze-Orchestrated Dynamic Windows," *Computer Graphics* **15**, 3 (August 1981).

[Bork79] Bork, A., *Textual Taxonomy*, Educational Technology Center, Physics Department, University of California, Irvine, CA, October 4, 1979.

[Bork81] Borkin, S.A. and Prager, J.M.,"Some Issues in the Design of an Editor-Formatter for Structured Documents," Technical Report, IBM Cambridge Scientific Center, September 1981.

[Bowm81] Bowman, W., and Flegal, B.,"ToolBox: A Smalltalk Illustration System," *BYTE*, vol. 6, no. 8, August 1981, pp. 369–376.

[Brow81] Brown University Computer Center, *User's Guide to the Brown CMS Editor*, Brown University, Box 1885, Providence, RI, April 1981.

[Burk80] Burkhart, H. and Nievergelt, J., "Structure-oriented Editors," Rep 38, Eidgenössische Technische Hochschule Zurich, Institute für Informatik, Zurich, Switzerland, May 1980.

[Burk81] Burkhart, H. and Stelovsky, J., "Towards an Integration of Editors," in *International Conference on Research Trends in Document Preparation Systems*, Lausanne, Switzerland, February 27-28, 1981, pp. 9–11.

[Bush45] Bush, V., "As We May Think," *The Atlantic Monthly*, vol. 176, no. 1, July 1945, pp. 101–108.

[Bush67] Bush, V., "Memex Revisted," in *Science Is Not Enough*, Vannevar Bush, Ed., William Morrow and Co., 1967, pp. 75–101.

[Card76] Card, S.K., Moran, P., and Newell, A., "The Manuscript Editing Task: A Routine Cognitive Skill," Report SSL-76-8, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, December 1976.

[Card78a] Card, S.K., "Studies in the Psychology of Computer Text Editing," Report SSL-78-1, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, August, 1978.

[Card78b] Card, S.K., English, W.K., and Burr, B.J., "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT," *Ergonomics* **21** (1978), 601–613.

[Card79] Card, S.K., Moran, P., and Newell, A., "The Keystroke-level Model for User Performance Time with Interactive Systems," Report SSL-79-1, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, March 1979.

[Carm69] Carmody, S., Gross, W., Nelson, T.H., Rice, D.E., and van Dam, A., "A Hypertext Editing System for the /360," in *Pertinent Concepts in Computer Graphics*, M. Faiman, J. Nievergelt, Ed., Illinois Press, March 1969, pp. 291–330.

[Cata79] Catano, J., "Poetry and Computers: Experimenting with the Communal Text," *Computers and the Humanities* **13** (1979), 269–275.

[Cham81] Chamberlin, D.D., King, J.C., Slutz, D.R., Todd, S.J.P., and Wade, B.W., "JANUS: An Interactive System for Document Composition," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 82–91.

[Cher81] Cherry, L., "Computer Aids for Writers," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 61–67.

[Cole69] Coleman, M., "Text Editing on a Graphic Display Device Using Hand-Drawn Proofreader's Symbols," in *Pertinent Concepts in Computer Graphics*, M. Faiman, J. Nievergelt, Ed., Illinois Press, March 1969, pp. 282–290.

[ComS67] Com-Share, Inc., *QED Reference Manual*, Ann Arbor, MI, 1967.

[Coul76] Coulouris, G.F., Durham, I., Hutchinson, J.R., Patel, M.H., Reeves, T., and Winderbank, D.G., "The Design and Implementation of an Interactive Document Editor," *Software - Practice and Experience* **6** (May 1976), 271–279.

[Cris65] Crisman, P.A., ed., *The Compatible Time Sharing System: A Programmer's Guide*, 2ndedition, MIT Press, 1965.

[Deut67] Deutsch, P. and Lampson, B., "An Online Editor," *Commun. ACM* **10**, 12 (December 1967), 793-799, 803.

[Digi78] Digital Equipment Corporation, *VAX-11 Text Editing Reference Manual*, Maynard, MA, August 1978.

[Donz75] Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J.J.,"A Structure Oriented Program Editor: A first step towards computer assisted programming," Technical Report, IRIA-LABORIA, France, April 1975.

[Donz80] Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B.,"Programming Environments Based on Structured Editors: the MENTOR experience," Technical Report, INRIA, France, May 1980.

[Dzid78] Dzida, W., Herda, S., and Itzfeldt, W.D., "User-Perceived Quality of Interactive Systems," *IEEE Trans. Software Engineering* **SE-4**, 4 (July 1978), 270–276.

[Elli80] Ellis, C. and Nutt, G., "Office Information Systems and Computer Science," *Comput. Surveys* **12**, 1 (March 1980), 27–60.

[Embl78] Embley, D.W., Lan, M.T., Leinbaugh, D.W., and Nagy, G., "A Procedure for Predicting Program Editor Performance from the User's Point of View," *International Journal of Man-Machine Studies* **10**, 6 (November 1978), 639–650.

[Embl81] Embley, D.W. and Nagy, G., "Behavioral Aspects of Text Editors," *Comput. Surveys* **13**, 1 (March, 1981), 33–70.

[Enge63] Engelbart, D.C.,"A Conceptual Framework for the Augmentation of Man's Intellect," in *Vistas in Information Handling*, P. Howerton, Ed., Spartan Books, Washington, DC, 1963, pp. 1–29.

[Enge68] Engelbart, D.C. and English, W.K.,"A Research Center for Augmenting Human Intellect," in *Proc. FJCC*, vol. 33, no. 1, AFIPS Press, Montvale, NJ, Fall 1968, pp. 395–410.

[Enge73] Engelbart, D.C., Watson, R., Norton, J.,"The Augmented Knowledge Workshop," in *Proc. NCC*, vol. 42, AFIPS Press, Montvale, NJ, June 4-8, 1973, pp. 9–21.

[Enge78] Engelbart, D.C.,"Toward Integrated, Evolutionary Office Automation Systems," in *Proc. Joint Engineering Management Conference*, Denver, CO, October 16-18, 1978, pp. 63–68.

[Engl67] English, W.K., Engelbart, D.C., and Berman, M.L., "Display-selection Techniques for Text Manipulation," *IEEE Trans. Hum. Factors. Electron.* **HFE-8**, 1 (March 1967), 5–15.

[Fajm73] Fajman, R., "WYLBUR: An Interactive Text Editing and Remote Job Entry System," *Commun. ACM* **16**, 5 (May 1973), 314–322.

[Feil80] Feiler, P.H. and Medina-Mora, R.,"An Incremental Programming Environment," Technical Report CMU-CS-80-126, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, April 1980.

[Fein81a] Feiner, S., Nagy, S., and van Dam, A., "An Integrated System for Creating and Presenting Complex Computer-Based Documents.," *Computer Graphics* **15**, 3 (August 1981).

[Fein81b] Feiner, S., Nagy, S., and van Dam, A., "Online Documents Combining Pictures and Text," *Proceedings of the International Conference on Research and Trends in Document Preparation Systems*, Lausanne, Switzerland (February 27-28, 1981).

[Fein82] Feiner, S., Nagy, S., and van Dam, A., "An Experimental System for Creating and Presenting Interactive Graphical Documents," *Transactions on Graphics* **1**, 1 (January 1982).

[Fins80] Finseth, C.A., "A Theory and Practice of Text Editors," Technical Memo 165, MIT Laboratory for Computer Science, Cambridge, MA, June 1980.

[Fins82] Finseth, C.A., "Managing Words: What Capabilities Should You Have with a Text Editor?," *BYTE*, vol. 7, no. 4, April, 1982, pp.242-282.

[Fole82] Foley, J.D. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

[Fras79] Fraser, C.W., "A Compact, Portable CRT-based Text Editor," *Software - Practice and Experience* **9**, 2 (February 1979), 121–125.

[Fras80] Fraser, C.W., "A Generalized Text Editor," *Commun. ACM* **23**, 1 (March 1980), 27–60.

[Fras81] Fraser, C.W., "Syntax-Directed Editing of General Data Structures," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 17–21.

[Frei78] Frei, H.P., Weller, D.L., and Williams, R., "A Graphics-Based Programming-Support System," *Computer Graphics* **12**, 3 (August 1978), 43–49.

[Furu82] Furuta, R., Scofield, J., Shaw, A., "Document Formatting Systems: Survey, Concepts, and Issues," *Comput. Surveys* **14**, 3 (September 1982).

[Gold79] Goldberg, A. and Robson, D., "A Metaphor for User Interface Design," in *Proc. Twelfth Hawaii International Conference on System Sciences*, vol. 6, no. 1, 1979, pp. 148–157.

[Gold80] Goldberg, A. and Robson, D., "Sharing Problems: Personal Computers as Interpersonal Tools," Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA. Keynote Address at the SIGSMALL/PC Symposium, Palo Alto, CA., September 1980.

[Gold82] Goldberg, A., "The Smalltalk-80 System: A User Guide and Reference Manual," Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, March 1, 1982.

[Gold83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, to appear 1983.

[Golc81] Goldfarb, C.F., "A Generalized Approach to Document Markup," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 68–73.

[Good81] Good, M., "Etude and the Folklore of User Interface Design," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 34–43.

[Good75] Goodwin, N.C., "Cursor Positioning on an Electronic Display Using Lightpen, Lightgun or Keyboard for Three Basic Tasks," *Hum. Factors* **17**, 3 (June 1975), 289–295.

[Gosl81] Gosling, J., "A Redisplay Algorithm," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 123–129.

[Gree80] Greenberg, B.S.,"Multics EMACS: An Experiment in Computer Interaction," in *Fourth Annual Honeywell Softare Conference*, March 1980.

[GRID82] GRiD Systems Corporation, *Compass Computer*, 2535 Garcia Avenue, Mountain View, CA, March 1982.

[GSPC79] GSPC, "Status Report of the Graphic Standards Planning Committee," *Computer Graphics* **13**, 3 (August, 1979).

[Habe79] Habermann, A.N., "An overview of the Gandalf Project," *Computer Science Research Review 1978-79*, Carnegie-Mellon University, Pittsburgh, PA (1979).

[Hamm81] Hammer, M., Ilson, R., Anderson, T., Gilbert, E.J., Good, M., Niamir, B., Rosenstein, L., and Schoichet, S.,"The Implementation of Etude, An Integrated and Interactive Document Production System," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 137–146.

[Hans68] Hansen, W.J.,"User Engineering Principles for Interactive Systems," in *Proc. FJCC*, vol. 39, AFIPS Press, Arlington, VA, Fall 1968, pp. 395–410.

[Hans71] Hansen, W.J.,"Creation of Hierarchic Text with a Computer Display," Report ANL7818, Argonne National Laboratory, July, 1971.

[Haye81] Hayes, P., Ball, E., and Reddy, R., "Breaking the Man-Machine Communication Barrier," *Computer* **14**, 3 (March 1981), 19–30.

[Haze80] Hazel, P., "Development of the ZED Text Editor," *Software - Practice and Experience* **10**, 1 (January 1980), 57–76.

[Hero80] Herot, C., Carling, R., Friedell, M., Kramlich, D., "A Prototype Spatial Data Management System," *Computer Graphics* **14**, 3 (July 1980).

[IBM67] International Business Machines, *Magnetic Tape Selectric Typewriter*, New York, 1967.

[IBM69] International Business Machines,"A Conversational, Context-Directed Editor," IBM Cambridge Scientific Center Report, Cambridge, MA, July 1969.

[IBM70] International Business Machines, *System/360 Administrative Terminal System*, 1970.

[IBM80] International Business Machines, *IBM Virtual Machine/System Product: System Product Editor User's Guide*, White Plains, NY, July 1980.

[Imag81] Image Data Products, Ltd., *The Image Data Tablet*, 1-4 Portland Square, Bristol, England, 1981.

[Iron72] Irons, E.T. and Djorup, F.M., "A CRT Editing System," *Commun. ACM* **15**, 1 (January 1972), 16–20.

[John75] Johnson, S.,"YACC - Yet Another Compiler-Compiler," Bell Labs, Murray Hill, NJ, 1975.

[Jong82] Jong, S.,"Designing a Text Editor? The User Comes First," *BYTE*, vol. 7, no. 4, April, 1982, pp. 284-300.

[Joy80a] Joy, W. and Horton, M.,"Ex Reference Manual – Version 3.1," , Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, September 16, 1980.

[Joy80b] Joy, W. and Horton, M., "An Introduction to Display Editing with Vi," , Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, Setember 16, 1980.

[Joy81] Joy, W. and Horton, M., "TERMCAP," in *UNIX Programmers Manual, Seventh Edition, Berkeley Release 4.1*, June, 1981.

[Kay77] Kay, A. and Goldberg, A., "Personal Dynamic Media," *Computer* **10**, 3 (March 1977), 31-43.

[Kell77] Kelly, J., "A Guide to NED: A New On-Line Computer Editor," R-2000-ARPA, Rand Corporation, Santa Monica, CA, July 1977.

[Kern75] Kernighan, B.W. and Cherry, L.L., "A System for Typesetting Mathematics," *Commun. ACM* **18**, 3 (March 1975), 182–193.

[Kern78a] Kernighan, B.W., "A Tutorial Introduction to the UNIX editor," Bell Laboratories, Murray Hill, NJ, September 28, 1978.

[Kern78b] Kernighan, B.W., "Advanced Editing on UNIX," Bell Laboratories, Murray Hill, NJ, August 4, 1978.

[Kern78c] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[Kern81] Kernighan, B.W., "PIC - A Language for Typesetting Graphics," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 92–98.

[Kern82] Kernighan, B.W. and Lesk, M.E., J. Nievergelt, G. Coray, J. Nicoud, A. Shaw, ed., *Document Preparation Systems*, North Holland, 1982.

[Knut79] Knuth, D.E., *TEX and Metafont: New Directions in Typesetting*, Digital Press, Bedford, MA, December 1979.

[LRG76] Learning Research Group, "Personal Dynamic Media," Technical Report SSL-76-1, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, March 1976.

[Lamp78] Lampson, B.W., "Bravo Manual," in *Alto User's Handbook*, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, November 1978, pp. 31–62.

[Lant79] Lantz, K. and Rashid, R., "Virtual Terminal Management in a Multiple Process Enviroment," in *Proc. 7th Symposium on Operating Systems Principles*, Pacific Grove, CA, December 10-12, 1979.

[Lant80] Lantz, K., "Uniform Interfaces for Distributed Systems," Computer Science Dept., University of Rochester, Rochester, NY, May 1980.

[Lede80] Lederman, A., "An Abstracted Bibliography on Programming Environments," Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA, June 1980.

[Ledg80] Ledgard, H., "The Natural Language of Interactive systems," *Commun. ACM* **23**, 10 (October 1980), 556–563.

[Lesk76] Lesk, M., "TBL – A Program to Format Tables," Technical Report 49, Bell Laboratories, Murray Hill, NJ, 1976.

[Lesk81] Lesk, M., "Another View," *Datamation*, vol. 27, no. 12, November, 1981, pp. 146.

[MacD80] MacDonald, N.H., Frase, L.T., and Keenan, S.A.,"Writer's Workbench: Computer Programs for Text Editing and Assessment," Bell Laboratories, Piscataway, NJ, 1980.

[MacL77] MacLeod, I.A., "Design and Implementation of a Display-oriented Text Editor," *Software - Practice and Experience* **7**, 6 (November 1977), 771–778.

[Maxe81] Maxemchuck, N.F., and Wilder, H.A., "Virtual Editing: I. The Concept," *Proceedings of the International Conference on Research and Trends in Document Preparation Systems*, Lausanne, Switzerland (February 27-28, 1981).

[McCa67] McCarthy, J., Dow, B., Feldman, G., and Allen, J.,"THOR–A Display Based Timesharing System," in *Proc. SJCC*, vol. 30, AFIPS Press, Arlington, VA, Spring 1967, pp. 623–633.

[Medi81] Medina-Mora, R. and Notkin, D.S.,"ALOE Users' and Implementor's Guide," Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, November, 1981.

[Meyr81] Meyrowitz, N. and Moser, M.,"BRUWIN: An Adaptable Design Strategy for Window Manager / Virtual Terminal Systems," in *Proc. 8th Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1981.

[Micr81] MicroPro, *WordStar User's Guide*, MicroPro International Corp., 1299 Fourth Street, Suite 400, San Rafael, CA, 1981.

[Mike81] Mikelsons, M.,"Prettyprinting in an Interactive Programming Environment," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 108–116.

[Mont82] Montgomery, E.B., "Bringing Manual Input Into the 20th Century: New Keyboard Concepts," *IEEE Computer* **15**, 3 (March 1982), 11–18.

[Morr81] Morris, J.M. and Schwartz, M.D.,"The Design of a Language-Directed Editor for Block-Structured Languages," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 28–33.

[Mukh80] Mukhopadhyay, A.,"A proposal for a hardware text processor," in *The Papers of the 5th Workshop on Computer Architecture for Non-Numeric Processing*, Pacific Grove, CA, March 11-14, 1980.

[Nass73] Nassi, I. and Shneiderman, B., "Flowchart Techniques for Structured Programming," *ACM SIGPLAN Notices* 8, 8 (August 1973).

[NBI81] NBI, Inc., *System 3000 Operator's Guide*, Boulder, CO, March 1981.

[Nels67] Nelson, T.H.,"Getting It Out of Our System," in *Information Retrieval: A Critical Review*, G. Schecter, Ed., Thompson Book Company, Washington, DC, 1967, pp. 191–210.

[Nels74] Nelson, T.H., *Computer Lib/Dream Machines*, Hugo's Book Service, Box 2622, Chicago, IL, 1974.

[Nels81] Nelson, T.H., *Literary Machines*, Box 128, Swarthmore, PA, 1981.

[Newm78] Newman, W.M.,"Markup," in *Alto User's Handbook*, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, November 1978, pp. 85–96.

[Newm79] Newman, W., and Sproull, R., *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.

[Norm81] Norman, D.A.,"The Trouble with UNIX," *Datamation*, vol. 27, no. 12, November, 1981, pp. 139–150.

[Notk79] Notkin, D.S. and Habermann, A.N.,"Software Development Environment Issues as Related to Ada," Carnegie-Mellon University, Pittsburgh, PA, 1979.

[Ossa76] Ossanna, J.,"NROFF/TROFF User's Manual," Technical Report 54, Bell Laboratories, Murray Hill, NJ, 1976.

[Pete80] Peterson, J.L., "Computer Programs for Detecting and Correcting Spelling Errors," *Commun. ACM* **23**, 12 (December 1980).

[Prus79] Prusky, J.N., *FRESS Resource Manual*, Department of Computer Science, Brown University, Box 1910, Providence, RI, 1979.

[Reid80a] Reid, B.K.,"A High-Level Approach to Computer Document Formatting," in *Proc. 7th Annual ACM Symposium on Programming Languages*, January 1980, pp. 24–30.

[Reid80b] Reid, B.K.,"Scribe: A Document Specification Language and its Compiler," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[Reid80c] Reid, B.K. and Walker, J.H., *Scribe User's Manual*, Third edition, Unilogic, Ltd, 605 Devonshire St., Pittsburgh, PA, 1980.

[Reid81] Reid, B.K. and Hanson, D.,"An Annotated Bibliography of Background Material on Text Manipulation," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 157–160.

[Reis81] Reiss, S.P., Lustig, M., and Medvene, L., *BB User's Guide*, Department of Computer Science, Brown University, Providence, RI, 1981.

[Rice71] Rice, D.E. and van Dam, A.,"An Introduction to Information Structures and Paging Considerations for On-line Text Editing Systems," in *Advances in Information Systems Science*, J. Tou, Ed., Plenum Press, New York, NY, 1971, pp. 93–159.

[Robt79] Roberts, T.,"Evaluation of Computer Text Editors," Report SSL-79-9, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, November 1979.

[Robg79] Robertson, G., McCracken, D., and Newell, A.,"The ZOG Approach to Man-Machine Communication," Technical Report CMU-CS-79-148, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Ocbober 1979.

[Sand78] Sandewall, E., "Programming in an Interactive Environment: the LISP experience," *Comput. Surveys* **10**, 1 (March 1978), 35–71.

[SeyJ81] Seybold, J., "Xerox's 'Star'," *The Seybold Report* **10**, 16 (April 27, 1981).

[SeyP78] Seybold, P.B., "Tymshare's Augment: Heralding a New Era," *The Seybold Report on Word Processing* **1**, 9 (October 1978), 1–16.

[SeyP79] Seybold, P.B., "The CPT 8000 and 6000 Word Processing Systems," *The Seybold Report on Word Processing* **2**, 1 (February 1979), 1–16.

[Smit82] Smith, D.C., Irby, C., Kimball, R., Verplank, B., and Harslem, E.,"Designing the Star User Interface," *BYTE*, vol. 7, no. 4, April, 1982, pp.242-282.

[Snee78] Sneeringer, J., "User-interface Design for Text Editing: A Case Study," *Software - Practice and Experience* 8 (1978), 544–557.

[Stal80] Stallman, R.M., "EMACS Manual for TWENEX Users," AI Memo 556, MIT Artificial Intelligence Laboratory, Cambridge, MA, August 17 1980.

[Stal81] Stallman, R.M., "EMACS: The Extensible, Customizable Self-Documenting Display Editor," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 147–156.

[Stro81] Stromfors, O. and Jonesjo, L., "The Implementation and Experiences of a Structure-Oriented Text Editor," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 22–27.

[Sufr81] Sufrin, B., "Formal Specification of a Display Editor," Technical Monograph PRG-21, Programming Research Group, Oxford University Computing Laboratory, June 1981.

[Suth63] Sutherland, I.E., "THOR–A Display Based Timesharing System," in *Proc. SJCC*, vol. 23, Spartan Books, Baltimore, MD, Spring 1963, pp. 329.

[Swin74] Swinehart, D., "Copilot: A MUltiple Process Approach to Interactive Programming Systems," Ph.D. Thesis, Stanford Artificial Intelligence Laboratory Memo AIM-230, Stanford University, Palo Alto, CA, July 1974.

[Symb81] Symbolics, Inc., *Symbolics Software*, 21150 Califa Street, Woodland Hills, CA, 1981.

[Teit81a] Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-directed Programming Environment," *Commun. ACM* **24**, 9 (September 1981).

[Teit81b] Teitelbaum, T., Reps, T., and Horwitz, S., "The Why and Wherefore of the Cornell Program Synthesizer," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 8–16.

[Teiw77] Teitelman, W., "A Display Oriented Programmer's Assistant," Report CSL-77-3, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, March 1977.

[Tesl79] Tesler, L., "Home Text Editing: A Tutorial," Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, 1979.

[Tesl81] Tesler, L., "The Smalltalk Environment," *BYTE*, vol. 6, no. 8, August 1981, pp. 90–147.

[Thim78] Thimbleby, H., "Character Oriented Text Editing," Computer Systems Laboratory, Queen Mary College, London University, London, England, November 1978.

[Tilb76] Tilbrook, D., "A Newspaper Page Layout System," M.Sc. Thesis, Department of Computer Science, University of Toronto, 1976.

[Toll65] Tolliver, B., "TVEDIT," Stanford Time-Sharing Memo No. 32, Dept. of Computer Science, Stanford University, Palo Alto, CA, 1965.

[Turb81] Turba, T.N., "Checking for Spelling and Typographical Errors in Computer-Based Text," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 51–60.

[vanD71a] van Dam, A., *FRESS (File Retrieval and Editing SyStem)*, Text Systems, Barrington, RI, July 1971.

[vanD71b] van Dam, A. and Rice, D.E., "On-line Text Editing: A Survey," *Comput. Surveys* **3**, 3 (September 1971), 93–114.

[vanW81] van Wyk, C.J.,"A Graphics Typesetting Language," in *Proc. ACM SIGPLAN/-SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 99–107.

[VIP69] VIP Systems, *VIPcom User's Guide*, Washington, DC, 1968.

[Walk81a] Walker, J.H.,"Personal communication," , July 1981.

[Walk81b] Walker, J.H.,"The Document Editor: A Supporting Environment for Preparing Technical Documents," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 44–49.

[Whit81] White, A.R.,"Pic – A C-based illustration language," , Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1981.

[Wild82] Wilder, H.A., and Maxemchuck, N.F., "Virtual Editing: II. The Interface," *Proceedings of SIGOA Conference on Office Automation Systems*, Philadelphia, PA (June 1982).

[Wood81] Woods, S.R.,"Z - The 95 Percent Program Editor," in *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, Portland, Oregon, June 1981, pp. 1–7.

[Xero82] Xerox Corporation, *8010 STAR Information System Reference Guide*, 1341 West Mockingbird Lane, Dallas, TX, 1982.